Objets conditionnels et objets inconnus

Christèle Faure

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Objets Conditionnels et Objets Inconnus

Christèle Faure

N° 2298

Juillet 94

PROGRAMME 2

Calcul symbolique,

programmation

et génie logiciel

ISSN 0249-6399

INRIA
SOPHIA ANTIPOLIS

Objets Conditionnels et Objets Inconnus

Christèle Faure*

Programme 2 — Calcul symbolique, programmation et génie logiciel

Projet SAFIR

Rapport de recherche n° 2298 — Juillet 94 — 62 pages

Résumé : Pour rapprocher la forme des calculs effectués par les systèmes de calcul formel, des calculs faits à la main, et faciliter ainsi l'utilisation des systèmes de Calcul Formel, nous avons introduit deux nouvelles classes d'expressions. Les premières qualifiées de conditionnelles sont des objets multi-valués, qui permettent de décrire $abs(n)$ comme l'objet "si $n \geq 0$ alors n sinon $-n$". Les secondes qualifiées d'inconnues (on en connaît le "type" mais pas la valeur) permettent d'exprimer "soit n un entier" grâce à un mécanisme de typage. Cette étude a donné lieu à une implémentation en `Axiom`.

Mots-clé : Calcul formel, Calcul Symbolique, Axiom

(Abstract: pto)

*. Travail financé par une bourse Post-Doc INRIA et le projet européen Esprit POSSO Basic Research Action 6846.

Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)

Conditional and unknown objects

Abstract: In order to improve the expressivity of Computer Algebra Systems, we have introduced two new classes of expression. First, we defined multi-valued expressions called *conditional* expressions, in order to express $abs(n)$ as the object "if $n \geq 0$ then n else $-n$". Then we have specified a second class called unknown (elements whose values are unknown, but whose type is known) in order to enable declaration such as "let n an integer". Examples of those two classes have been implemented in `Axiom`.

Key-words: Computer algebra, Symbolic Computation, `Axiom`

Chapitre 1

Introduction

Le Calcul Formel se différentie du Calcul Numérique essentiellement par le mode de manipulation des symboles. En effet, le calcul numérique n'utilise les symboles que comme des variables lors de la programmation, c'est à dire pour ranger un résultat intermédiaire. Un programme numérique ne rend jamais un résultat sous forme de symbole mais sous forme de nombre ou de tableau de nombres. Les Systèmes de Calcul Formel utilisent les symboles non seulement comme variables de programmation, mais aussi comme des "variables muettes" dans les polynômes.

Mais il existe une autre classe de symboles fréquemment utilisés en mathématique dont il n'existe pas d'équivalent dans les Systèmes. En effet, les énoncés mathématiques commencent souvent par des phrases de la forme "soit **p** un polynôme" ou "soit **n** un entier"... Dans ces énoncés, les symboles **p**, **n** sont des symboles qui ne peuvent être traités comme des variables de polynômes. Ces symboles ne sont associés ni à une "valeur" comme les variables de programmation, ni à une "position" comme les variables muettes, mais à un "type".

Nous appellerons cette classe d'expressions:

les objets $\mathcal{INCONNUS}$.

Dans les systèmes actuels, cette catégorie d'objets n'existe pas, ce qui amène souvent des divergences entre les résultats obtenus et ceux attendus par l'utilisateur. Par exemple, l'expression **n+m** peut avoir autant de sens qu'on veut, elle peut signifier aussi bien la somme de deux matrices, la concaténation de deux listes ... Par la déclaration préalable "soient **n,m** deux entiers" elle n'a plus que le sens de somme de deux entiers. La définition d'inconnues en préambule à un calcul a donc pour but de préciser ce calcul.

On définit maintenant un ensemble précis d'inconnues: les inconnues entières, muni des opérations d'anneau, et de l'opérateur *abs*.
Si n et m sont des inconnues entières, le système peut calculer $2 * abs(n+m) - abs(2*(n+m))$ comme une inconnue entière. Mais cette information de type n'est pas suffisante, on voudrait aller plus loin et pouvoir "calculer" cette expression en 0 comme on le fait à la main.
Cette transformation en 0 est rendue possible grâce à la définition de l'opérateur *abs* sur les entiers. On peut toujours définir le calcul de *abs* sur un entier X comme un calcul multiple dont le résultat est:

$$\begin{pmatrix} X<0 & -X \\ X=0 & 0 \\ X>0 & X \end{pmatrix}.$$

Ces calculs multiples sont décrits par des objets multi-valués appelés :

les objets $\mathcal{CONDITIONNELS}$.

Si on étend les opérateurs valables sur les inconnues entières $+, *, -, abs...$ aux inconnues entières conditionnelles, le calcul de $2 * abs(n+m) - abs(2*(n+m))$ se fait comme :

$$2 * \begin{pmatrix} n+m<0 & -(n+m) \\ n+m=0 & 0 \\ n+m>0 & n+m \end{pmatrix} - \begin{pmatrix} 2*(n+m)<0 & -2*(n+m) \\ 2*(n+m)=0 & 0 \\ 2*(n+m)>0 & 2*(n+m) \end{pmatrix}$$
$$\begin{pmatrix} n+m<0 & 0 \\ n+m=0 & 0 \\ n+m>0 & 0 \end{pmatrix}$$

On peut remarquer à partir de cet exemple simple, que les calculs sur les conditions se compliquent rapidement. Par exemple, il faut pouvoir déduire que $n+m<0 \wedge 2*(n+m)=0$ est toujours faux, ou que $n+m<0 \wedge 2*(n+m)<0$ est équivalent à $n+m<0$.

Dans cette introduction, nous avons essayé de faire apparaître la nécessité de définir des objets conditionnels, ainsi que des objets inconnus sur des exemples très simples.
Dans le premier chapitre de ce document, nous allons montrer comment ces besoins s'expriment pour des utilisateurs de Systèmes de Calcul Formel à travers des messages tirés des **News**.
Dans le deuxième chapitre nous allons définir les objets conditionnels, les spécifier et en présenter une application. Enfin le troisième chapitre contient la définition des inconnues, leur spécification en `Axiom` et présente une classe d'inconnues : les inconnues entières.

Nous présentons en annexe de ce document, les définitions des Catégories ainsi que de quelques Domaines décrits dans les chapitres précédants.

1.1 Exemples de besoins utilisateurs

Nous avons extrait certains messages des **News** décrivant les problèmes rencontrés par les utilisateurs de **Systèmes de Calcul Formel**.
L'analyse de ces messages nous a permis de mettre à jour le besoin d'objets multi-valués qualifiés de **conditionnels** car chaque valeur est liée à une condition qui décrit son domaine de validité. Le besoin de "parler" d'un objet grâce à son nom après lui avoir donné un type mais aucune valeur, c'est à dire l'utilisation d'objets **inconnus** émerge aussi de ces messages.

Ce chapitre présente ces problèmes utilisateur ainsi que les solutions que nous pouvons y apporter grâce à l'utilisation des objets conditionnels ou des objets inconnus.

Message 1 *I don't know whether this matter has been brought up before, but I would like to know why Mathematica gives :*

```
Integrate[x^n,x]
x^(n+1)/(n+1)
```

which of course is wrong for n=-1, instead of returning an object of the kind :

```
If[ n==-1, Log[Abs[x]], x^(n+1)/(n+1)].
```

Generally I would like to know how you can specify for instance that a variable is real, or that a whole term (say `x^2+1`*) is greater than another, ... You can't (?) simply say* `Real[x]=True`*, because "Real" is protected, and if you unprotect it, you could do much harm, if for instance you type* `Real[x_]=True`*... Is there a safe and easy way to do such things in Mathematica?*

PS: The above said does of course not only apply to Mathematica, but also other symbolic math programs. As someone else pointed out, if you have an assume() function (as for instance in MAPLE) you always got the problem to tell what particular assumption the program would need to evaluate your input...

Please reply to wsi@vexpert.dbai.tuwien.ac.at

Analyse du message 1 *Le premier problème, concernant l'intégration, peut être simplement résolu si le résultat de l'appel de la fonction* `Integrate` *est une expression conditionnelle telle que :*

$$\begin{pmatrix} n < -1 & \rightarrow & x^{n+1}/(n+1) \\ n = -1 & \rightarrow & Log[Abs[x]] \\ n > -1 & \rightarrow & x^{n+1}/(n+1) \end{pmatrix}$$

Il est bien évident que si on autorise l'existence de tels objets dans un système, il faut définir toutes les opérations valides sur ces objets, telles que la somme, le produit, la valeur absolue ...

Le second problème peut être résolu en disant que `x` *est un symbole typé (*`Real`*) mais sans valeur, donc une inconnue. La définition des inconnues réelles semble répondre à son problème. En effet, si* `x` *est une inconnue réelle, et si les opérateurs* `^,+` *sont définis sur ces inconnues* `x^2+1` *est alors lui aussi une inconnue réelle.*

Remarque 1 *On peut supposer que si cet utilisateur ne veut pas utiliser les expressions générales, c'est qu'il a besoin de définir certains opérateurs sur ces inconnues qui ne sont pas définis de la même façon sur les vrais réels du système (*`Real`*).*

Message 2 *I would like to put in a package a line that constructs a list (of length m given, let's say, by the user) of series (of the domain UTS, let's say) in which elements have not been given values.*
The way I thought it would be was :
`u:List(UTS(k,x,0$k)):=[u.i for i in 1..m]` *where k is a field.*

The u.i have no values here. I just want them to have the type `UTS(k,x,0$k)`*, cause I'll have to use them in a system of differential equations. I tried to declare u only as a List of series,* `u:=List(UTS(...))`*, which has worked, but I can't use the u.i as abstract series.*

François Huard
huarf00@dmi.usherb.ca
Département de math. info.
Université de Sherbrooke
Québec, Canada.

Analyse du message 2 *Cet utilisateur veut pouvoir définir des objets de nom* `u.i` *de type série n'ayant pas de valeur, et ceci afin de les utiliser dans des calculs en lieu et place de véritables séries de type* `UTS(k,x,0$k)`*. L'utilisation de séries inconnues permettrait de résoudre sont problème. Le fait de vouloir les nommer comme les éléments d'une liste est à découpler de ce problème.*

Message 3 *I would like to be able to do some matrix manipulations at a symbolic level, without ever explicitly assigning a numeric dimension. An example of what I would like to be able to do could look something like this :*

```
assume(m,posint);
assume(n,posint);
A:=matrix(m,n);
B:=matrix(m,n);
multiply(B,B);
error: matrix dimensions incorrect;
C:=multiply(A,transpose(B));
                                AB';
D:=inverse(A);
                                A^-1;
F:=multiply(A,C);
                                 B';
```

Has anybody developed such a package? Any pointers would be appreciated.

Thanks,
Phil West
Georgia Tech Research Institute
phil.west@gtri.gatech.edu

Analyse du message 3 *Ici encore la définition d'inconnues matricielles peut résoudre le problème. On peut remarquer que dans ce cas, les inconnues de bases (*`A,B`*) sont non seulement décrites par leur nom (*`A,B`*), mais aussi par leur taille* `n,m` *et que suivant ces tailles certaines opérations sont valides ou non.*

Chapitre 2

Objets Conditionnels

Dans ce chapitre, nous allons analyser la notion d'objet conditionnel, rendre effective cette analyse (en `Axiom`), exhiber un exemple de telle classe d'objets (expressions conditionnelles) ainsi qu'une utilisation de cette classe.

2.1 Définition

- Qu'est-ce qu'un objet conditionnel?
  C'est une expression exp qui peut avoir plusieurs valeurs suivant des conditions. Ce qui peut s'exprimer par "sous la condition C_1, l'expression vaut exp_1, sous la condition C_2, l'expression vaut exp_2 ..." ou de façon plus visuelle (comme une définition par morceaux) par:

$$exp = \left| \begin{array}{l} exp_1 \text{ si } C_1 \\ exp_2 \text{ si } C_2 \\ exp_3 \text{ si } C_3 \\ \cdots \end{array} \right.$$

  Nous appelerons $exp_1, ...$ les valeurs de cette objet, et $C_1, ...$ les conditions.

- Que dire des conditions?
  Si on regarde un objet conditionnel comme une définition de fonction par morceaux, les conditions décrivent un unique ensemble. Mais sur cet ensemble on peut imaginer une multitude de fonctions de "fractionnement". Par exemple sur les entiers ont peut penser aux fonctions de comparaison $(<, >, =)$ ou à des définitions de congruence.

- Que dire des valeurs?
  Les valeurs sont bien entendu toutes de même type car une fonction décrite par morceaux est définie d'un domaine de départ vers un unique domaine d'arrivée.


Est-ce que ces objets sont **complets** ou **partiels**, c'est à dire les conditions de chacun d'entre eux recouvrent-t'elles entièrement le domaine de définition.
Par exemple, l'expression suivante de $f(n)$ est partielle,

$$\begin{array}{rcccl} f(n) & = & n & \text{si} & 0 \leq n \\ & & -n & \text{si} & n < -3, \end{array}$$

alors que cette dernière est complète

$$f(n) = \begin{array}{lll} n & \text{si} & 0 \leq n \\ n+3 & \text{si} & -3 \leq n < 0 \\ -n & \text{si} & n < -3. \end{array}$$

Définition 1 *Un objet formé de conditions et de valeurs est complet si et seulement si la disjonction de ses conditions est toujours vraie.*

Définition 2 *Un objet formé de conditions et de valeurs est cohérent si et seulement si sous une condition, il ne possède qu'une seule valeur. C'est à dire que si quand deux conditions C_i, C_j sont compatibles ($C_i \wedge C_j$ ne se réduit pas à faux), les deux valeurs V_i, V_j sont semblables.*

Définition 3 *Un objet formé de conditions et de valeurs est un objet conditionnel si et seulement si il est complet et cohérent.*

Remarque 2 *Il faut donc que toutes les fonctions définies pour un domaine conditionnel conservent cette complétude.*

Choix 1 *Les conditions appartiennent à un même domaine sur lequel un ensemble homogène d'opérateurs de "fractionnement" est défini.*

2.2 Spécification

Nous allons utiliser la capacité descriptive du langage d'`Axiom` pour spécifier les objets conditionnels. La définition précédente permet de dire qu'un domaine conditionnel se défini à partir d'un domaine de conditions `Cond` et d'un domaine de valeurs `Val`.

2.2.1 Quelques précisions sur `Axiom` et son langage

`Axiom` est un Système de Calcul Formel qui peut être utilisé, soit comme une calculatrice programmable, soit comme un langage de programmation modulaire orienté "structure de donnée pour les mathématiques". C'est, en gros, un système constitué d'une bibliothèque et d'un compilateur permettant de créer de nouveaux fichiers dans la bibliothèque.
Dans ce système, l'utilisateur est parfaitement à même de définir de nouveaux objets mathématiques, ainsi que les opérations valides sur ces objets à travers le langage de programmation. Ceci est fondamentalement différent des systèmes classiques. Nous allons ici nous attacher surtout à décrire cette vision langage de programmation.

Un ensemble d'objets ainsi que les opérations qui permettent de les manipuler est appelé `domain` en `Axiom`. Pour construire un domaine, il est possible soit d'utiliser un `foncteur` standard, soit de créer un nouveau foncteur de domaines (voir [JS92a]). Pour créer ce foncteur, il faut tout d'abord spécifier les domaines qu'il engendrera en terme d'appartenance à des ensembles plus abstraits appelés `categories`. Les `categories` sont organisées les unes par rapport aux autres en fonction de propriétés algébriques pour les catégories "mathématiques" et structurelles pour les catégories "informatiques" en un graphe hiérarchique (voir les pages intérieures de la couverture de [JS92b]).

Il est bien sûr possible d'ajouter une catégorie à la hiérarchie si ce domaine appartient à une nouvelle classe de domaines en créant une fonction [1] (voir [JS92c]).

Voici le modèle général de définition que nous allons suivre dans la suite :

```
X (param1,param2 ...): LHS == VHS where

 param1 : T1
 param2 : T2
 ...

 LHS ==>
 VHS ==>
```

Ceci définit la fonction de nom `X` ayant pour paramètres `param1,param2 ...` de types respectivement `T1,T2,....`
Le signe `==>` permet de définir une macro.
Si `LHS` s'expense en `Category`, alors `X` défini une catégorie, et `VHS` (noté `Exports`) est alors le type de la catégorie.
Sinon `X` est un foncteur de domaines, et `LHS` (noté `Exports`) est le type de ces domaines, et `VHS` (noté `Implementation`) en est l'implémentation.

Un `Exports` est défini comme un `Join` de types prédéfinis (pouvant se réduire à un unique type) suivi du mot clef `with` et des types des fonctions supplémentaires de la forme
`toto :(T1,...Tn) -> Tn1+` où `toto` est le nom de la nouvelle fonction, `(T1,...Tn)` sont les type d'entrée de ses arguments et $Tn+1$ est son type de sortie.
Une `Implementation` est constituée d'une implémentation prédéfinie (pouvant être vide) suivie du mot clef `add` et de définitions de fonctions de la forme `toto (x1,...xn) == ...` où $x1,...xn$ sont ses paramètres d'entrée.

Remarque 3 *Dans une définition de domaine de catégorie, le symbole* `$` *représente le type défini.*

`Axiom` *autorise l'associations d'abréviations aux noms de foncteurs de catégories et de constructeurs de domaines par l'opérateur* `)abbrev`. *Ces abréviations, répertoriées page 44, seront utilisées dans le texte pour racourcir les types.*

2.2.2 Catégorie de domaines de conditions

Les conditions sont construitent par négation, conjonction, disjonction de conditions élémentaires. On ne peut rien dire sur les conditions él'ementaires, si ce n'est que ce sont des fonctions à valeur booléenne. Ces opérations effectuées sur les conditions élémentaires appartiennent à la logique booléenne (and, or, not, ...), nous avons donc créé une catégorie des domaines de conditions exportant ces opérateurs.

Cette catégorie appelée
`ExtendedBooleanCategory` (abbrégé en `EBCAT`) est une généralisation du type du domaine `Boolean`, elle exporte donc les opérateurs booléens classiques "and,or ...".


A la lecture, `Axiom` transforme les expressions de la forme "a and b" en "if a then a else b". Il n'est donc pas possible d'utiliser ces mêmes symboles pour signifier le "et" entre deux booléens conditionnels car le "if" demande un vrai booléen. Les expressions contenant les opérateurs or, not, true, false sont aussi

1. Cette fois ci, il ne s'agit pas d'un foncteur car les catégories peuvent être paramétrées.

transformées, nous avons donc changé les noms des opérateurs de base : and, or, not, true, false, ... en : `And`, `Or`, `Not`, `True`, `False`...

La définition de cette catégorie est donc :

```
EBCAT: Category == Exports where

 Exports ==> SetCategory with
    True  : -> $
    False : -> $
    Not   : $ -> $
    And   : ($,$) -> $
    Or    : ($,$) -> $
    ...
```

Il nous a fallu exporter une fonction supplémentaire qui permet de comparer syntaxiquement de booléens étendus appelée `inf?`. Nous n'avons pas donné aux domaines de type `EBCAT` la propriété d'`OrderedSet` car cet ordre n'est pas sémantique, mais purement syntaxique et sera utilisé comme tel. (Les `EBCAT` pourraient par contre faire parti d'une catégorie `ConditionalOrderedSet`.)

Comme nous l'avons vu précédemment, les objets conditionnels sont cohérents et complets. La qualité des calculs sur ces objets dépend donc de celle des calculs sur les conditions, en particulier il est fondamental de pouvoir dire qu'une condition est fausse (vraie) quels que soient les valeurs de ses paramètres. Nous avons donc ajouté au calcul syntaxique des booléens étendus ($P1 > 0 \wedge P1 > 0 \rightarrow P1 > 0$) un mécanisme général de calcul "sémantique" modulo des hypothèses.

Exemple 1 *Si on fait un calcul purement syntaxique, l'expression suivante :*

$$positive(n(m-1)) \vee negative(n(m-1)) \vee null(n) \vee null(m-1)$$

ne peut être simplifiée à vrai.
Par contre, si on déclare que :

$$n(m-1) = 0 \Leftrightarrow n = 0 \vee m - 1 = 0,$$

l'expression sera simplifiée en

$$True.$$

Les hypothèses à partir desquelles sont effectués les calculs sont de deux sortes[2] :

- prédéfinies dans le système
$$\forall i \quad P_i > 0 \wedge P_i < 0 \Leftrightarrow False$$
- déclarées par l'utilisateur (ou un autre mécanisme de calcul)
$$n(m-1) = 0 \Leftrightarrow n = 0 \vee m - 1 = 0,$$

Nous utilisons une méthode de Robinson pour résoudre les conditions à des constantes, en considérant les hypothèses prédéfinies implicitemant et les autres explicitemant.

Ce calcul pouvant être lourd n'est pas effectué automatiquement, mais doit être demandé par l'utilisateur par l'intermédiaire de la fonction `simplify`.

2. Les exemples sont donnés dans le cas réèl.

La catégorie `EBCAT` est donc étendue en une nouvelle catégorie appelée `ExtendedBooleanModAssertionsCategory` (abbrégé en `EBMACAT`) définie par :

```
EBMACAT A : Category == Exports where

  A : SetCategory

  Exports ==> EBCAT with
   assertions                : -> List A
   asserted                  : -> $
   assert                    : ($,$) -> A
   clearAssertions           : -> List A
   computeModAssertions      : $ -> $
   contradictoryToAssertions : $ -> Boolean
   simplify                  : $ -> $
```

Les booléens étendus peuvent être considérés sous forme disjonctive ou conjonctive; chaque forme optimisant une classe de calculs. Dans notre cas, il est préférable de regarder ces expressions sous forme disjonctive (c.a.d disjonctions de conjonctions de conditions élémentaires) puisque les expressions conditionnelles sont elles même des disjonctions. Pour rendre cette manipulation claire, nous avons défini deux catégories: l'une pour les domaines de conditions élémentaires `ElementaryConditionCategory` (abbrégé en `ECCAT`), l'autre pour les booléens conditionnels manipulés sous forme disjonctive `DisjunctiveConditionCategory` (abbrégé en `DCCAT`). Voici leurs définitions :

```
ECCAT : Category == Exports where

 Exports ==> SetCategory with
   inf? : ($,$) -> Boolean
   contradictory? : ($,$) -> Boolean
   redundant? : ($,$) -> Union (Integer,"failed")
   Not : $ -> List List $

DCCAT E : Category == Exports where

 E : ECCAT

 Conjunction ==> List E           -- implicit 'and' [] = true
 Disjunction ==> List Conjunction -- implicit 'or'  [] = false

 Exports ==> EBCAT with
   destruct     : $ -> Disjunction
   construct    : E -> $
   construct    : Disjunction -> $
```

2.2.3 Catégorie de domaines d'objets conditionnels

Chaque opération exportée par un domaine (d'objets) conditionnel doit construire exclusivement des objets complets (voir définition page 8). Pour rendre cette vérification claire, chaque domaine conditionnel est doublé d'un domaine d'objets partiels dans lequel sont effectuées toutes les manipulation qui ne conservent pas la complétude, mais conserve la cohérence. Tout domaine partiel est utilisé à l'intérieur d'un domaine conditionnel, et n'est donc pas visible.

Nous avons défini une catégorie des domaines conditionnels complets, et une pour les domaines conditionnels partiels qui prennent toutes deux en arguments un domaine de conditions de `ExtendedBooleanCategory` et un domaine de valeurs de `SetCategory`.

Catégorie des domaines d'objets partiels

Les fonctions exportées par les domaines partiels construisent des listes de couples condition/valeur, elles sont donc à rapprocher des fonctions du type `IndexedAggregate`. Mais il nous a semblé plus simple de définir cette catégorie comme appartenant à `setCategory` que de donner un sens à toutes les fonctions de cette catégorie prédéfinie.

La définition de la catégorie `PartialConditionalCategory` (abbrégé en `PCCAT`) est donc :

```
PCCAT (C,V):Category == Exports where

 C : EBCAT,
 V : SetCategory

 Exports ==> SetCategory with
   empty  : -> $
   concat : ($,C,V) -> $
        ...
```

Les fonctions `concat` vérifient que l'objet partiel qu'elle construisent est cohérent, et provoque une erreur si ce n'est pas le cas.

Catégorie des domaines d'objets complets

Les fonctions communes à tous les domaines conditionnels permettent d'extraire d'un objet conditionnel les informations pertinentes :

1. la liste des valeurs : `values`,
2. la liste des conditions : `conditions`,
3. la valeur correspondant à une condition : `whichValue`,
4. la condition correspondant à une valeur : `whichCondition`.

ou de construire :

1. un objet conditionnel à partir d'un objet partiel et réciproquement : `coerce`,
2. un objet conditionnel à partir d'une condition et de deux objets conditionnels : `ConditionalValue`.

La définition de la catégorie `ConditionalCategory` (abbrégé en `CCAT`) est donc :

```
CCAT (C,V):Category == Exports where

  C : EBCAT
  V : SetCategory

  Exports ==> SetCategory with
        values : $ -> List V
        whichValue : ($,C) -> V
        ...
```

Foncteur de domaines d'objets conditionnels

Un domaine conditionnel est défini comme l'extension de son domaine de valeurs, les mêmes opérations se retrouvent donc dans les deux.
Pour exprimer cela, on peut dire en `Axiom` que les deux domaines appartiennent à la même catégorie, donc ont même type. La définition du constructeur général de domaines[3] conditionnels pourrait donc être :

```
ConditionalY (Cond:X,Val:Y):Category == Exports where

 Exports ==> Join (Y,CCAT (Cond,Val))
```

Remarque 4 *Certaines opérations n'ont aucun sens sur des objets multivalués tels que les objets conditionnels, et ne sont donc pas étendus. On ne prend pas le type maximum de* `Val`*, mais un "morceau" significatif de son type.*

2.3 Implémentation

2.3.1 Représentation

Nous avons défini deux foncteurs de domaines définissant progressivement le domaine de condition voulu, et implémentant des simplifications spécifiques à chaque niveau de manière à ce que les calculs puissent toujours être contrôlés par l'utilisateur.

Voici le foncteur de domaines de conditions vues comme des disjonctions de conditions élémentaires appelé `DisjunctiveCondition` (abbrégé en `DC`) :

```
DC E : Exports == Implementation where

 E : ECCAT

 Exports ==> DCCAT E
```

Les conditions sont calculées automatiquement modulo les contradictions et les redondances testées grâce aux opérations sur les conditions élémentaires.

Le foncteur `DisjunctiveConditionModEquivalences` (abbrégé en `DCME`) de domaines de conditions vues comme des disjonctions de conditions élémentaires calculées modulo équivalences est défini ainsi :

```
DCME E: Exports == Implementation where

  E : ECCAT

  Equivalence ==> Record (lhs:$,rhs:$)

  Exports ==> Join (DCCAT E,EBMACAT Equivalence)
```

La fonction simplify calcule son argument modulo les équivalences déclarées et doit être appelée explicitement pour effectuer ces transformations. En effet, ces calculs peuvent être longs et ne peuvent donc être faits

3. Voir [JS92a] pour des détails sur cette notion.

automatiquement lors des calculs.
Ceci permet aussi lors d'une implémentation utilisant ce type d'effectuer les simplifications à l'endroit désiré, par exemple à la fin d'un "gros" calcul et non à chaque "petit" calcul.

Un domaine de condition est donc représenté comme l'un de ces deux domaines suivant qu'on veut ou non faire des calculs modulo équivalences.

Les objets multi-valués sont représentés comme des listes de couples condition-valeur, ce qui se traduit en `Axiom` par le type `List Record (cond:C,val:V)`. Nous n'utilisons qu'une seule représentation pour les objets d'un domaine partiel et de son domaine complet associé, mais pour `Axiom` ils restent entièrement discernables.
Les foncteurs de domaines d'objets partiels `PartialConditional` (abbrégé en `PC`) et conditionnels `Conditional` (abbrégé en `CO`) peuvent alors être définis comme suit :

```
PC (C,V):Exports == Implementation where

 C : EBCAT
 V : SetCategory

 Term ==> Record (cond:C,val:V)

 Exports ==> PCCAT (C,V)
 Implementation ==> add
   Rep := List Term

CO (C,V):Exports == Implementation where

 C : EBCAT
 V : SetCategory

 Term ==> Record (cond:C,val:V)

 Exports ==> CCAT (C,V)
 Implementation ==> add
   Rep := List
```

2.3.2 Remarque

On peut remarquer que pour effectuer une opération sur des objets conditionnels, il faut effectuer un type d'opérations sur les conditions et un autre sur les valeurs.

Exemple 2 *Soit $abs(n)$ et m représentés comme deux objets conditionnels, voici le calcul de $abs(n) + m$:*

```
((n>0) -> n)                      ((n>0) -> n + m)
((n=0) -> 0)      +   (-> m)  =   ((n=0) -> m)
((n<0) -> - n)                    ((n<0) -> - n + m)
```

et le calcul de $abs(n) < m$:

```
                                  (And (n<0,-n-m<0) -> True)
                                  (And (n<0,-n-m=0) -> False)
                                  (And (n<0,-n-m>0) -> False)
((n<0) -> -n)                     (And (n=0,-m<0) -> True)
```

```
((n=0) -> 0)      <   (-> m)  =  (And (n=0,-m=0) -> False)
((n>0) -> n)                     (And (n=0,-m>0) -> False)
                                 (And (n>0,n-m<0) -> True)
                                 (And (n>0,n-m=0) -> False)
                                 (And (n>0,n-m<0) -> False)
```

Nous avons classé les fonctions qui effectuent de tels calculs en deux catégories, suivant les liens qu'elles impliquent entre calculs sur les conditions et sur les valeurs :

1. les fonctions pour lesquelles les calculs sur les conditions et les valeurs sont indépendants (tel que dans le calcul de $abs(n) + m$),
2. les fonctions nécessitant un calcul sur les valeurs qui ajoute des conditions (tel que dans le calcul de $abs(n) < m$).

Nous avons dégagé deux modèles de telles fonctions génératrices: une unaire et une binaire. Soit `V1` (respectivement `V2`) un domaine, `PCV1` (`PCV2`) son domaine conditionnel partiel associé et `CV1` (`CV2`) son domaine conditionnel associé, voici les types de ces fonctions :

1. `apply1 : (V1 -> V2,CV1) -> CV2`
   `apply2 : ((V1,V1) -> V2,CV1,CV1) -> CV2`
2. `apply1 : ((V1,C) -> PCV2,PCV1) -> PCV2`
   `apply2 : ((V1,C,V1,C) -> PCV2,PCV1,PCV1) -> PCV2`

Les fonctions `apply1,apply2` de la première classe construisent des objets complets. Ces fonctions sont définies dans le package `ConditionalFunctions` (abbrégé en `CF`). Quand les calculs sont internes (c.a.d `CV1` = `CV2`), il est plus simple de les définir (moins de conversions entre `PCV1` et `CV1`) dans `CV1`, nous avons donc ajouté ces définitions à la catégorie `CCAT`.

```
CF (C,V1,V2,CV1,CV2) : Exports == Implementation where

 C   : EBCAT
 V1  : SetCategory
 V2  : SetCategory
 CV1 : CCAT (C,V1)
 CV2 : CCAT (C,V2)

 Exports ==> with
  apply1 : ((C,V1) -> CV2,CV1) -> CV2
  apply2 : ((C,V1,C,V1) -> CV2,CV1,CV1) -> CV2
```

Les fonctions `apply1,apply2` de la deuxième classe construisent des objets partiels, nous avons défini un package qui exporte ces fonctions applicatives, nommé:
`PartialConditionalFunctions` (abbrégé en `PCF`).

```
PCF (C,V1,V2) : Exports == Implementation where

 C  : EBCAT
 V1 : SetCategory
 V2 : SetCategory
```

```
PCV1 ==> PC (C,V1)
PCV2 ==> PC (C,V2)

Exports ==> with
 apply1 : ((C,V1) -> PCV2,PCV1) -> PCV2
 apply2 : ((C,V1,C,V1) -> PCV2,PCV1,PCV1) -> PCV2
```

2.4 Exemple : Expressions (générales)

2.4.1 Spécification

Nous avons défini le constructeur `ConditionalExpression` (abbrégé en `CEX`) en prolongeant le constructeur d'expressions `Expression`.

Expressions conditionnelles

Le foncteur de domaines d'expressions prend en argument un domaine `R` de type `OrderedSet`. Le type algébrique du domaine qu'il construit dépend du type de `R`, par exemple si `R` est un `SemiGroup`, `Expression R` est un `Monoid`.

Nous avons conservé aux domaines d'expressions conditionnelles ce même type algébrique, ainsi que les facilités de conversions, de rétractions qui sont importantes lors de l'utilisation d'un tel domaine.

Par contre nous avons supprimé des fonctions exportées,

- `paren, box, quote, height`, ... qui servent à manipuler les "images à l'écran" des expressions qui ne nous semblent pas être du même ordre,
- `<` qui pose un problème de sémantique car il correspond parfois à une comparaison numérique et parfois à une comparaison syntaxique,
- `eval, subst, patternMatch` pour lesquelles il faut savoir si :
  1. on veut transformer l'expression en remplaçant les noyaux par des `CEX` ou des `Expression`?
  2. on veut remplacer les noyaux à la fois dans les conditions et les valeurs ou seulement dans les valeurs et ajouter l'équation dans les conditions (contraintes) ...?
- `product, summation, integral` (sur les segments) car elles utilisent les fonctions précédentes,
- `reducedSystem`, car on ne veut pas pour l'instant construire de matrices d'expressions conditionnelles.

Les fonctions de `Expression` qui ont été étendues, excepté `abs`, ne construisent pour l'instant que des constantes de `CEX`, donc des objets sans condition.
Pour que les expressions de la forme $abs(e)$ soient transformées en "si $e > 0$ alors e, si $e = 0$ alors 0, sinon $-e$, il faut à la fois que l'opérateur *abs* construise un objet conditionnel, mais aussi que la fonction de conversion d'une expression en une expression conditionnelle analyse son argument et recherche les opérateurs *abs*. Sinon, en utilisant `abs` on aurait obtenu une expression conditionnelle bien construite, et par conversion d'une expression conditionnelle constante.

Pour que le calcul sur les expressions conditionnelles contenant π puissent être similaire à celui effectué sur les expressions, nous avons défini un package de coercion entre π et les expressions conditionnelles appelé `ConditionalPiCoercionPackage` (abbrégé en `CPICPAC`) :

```
CPICPAC R : Exports == Implementation where

  R : Join (OrderedSet,IntegralDomain)
  E ==> Expression R

  Exports ==> with
    coerce: Pi -> CEX R
```

Conditions sur les expressions

Les conditions élémentaires nécessaires à la définition de l'opérateur `abs` sont des comparaisons à zéro d'expressions.

Nous avons défini une catégorie `ComparisonsToZeroCategory` (abbrégé en `CTZCAT`) des domaines de comparaisons à zéro d'objets généraux (de type `V`) comme suit :

```
CTZCAT (V:SetCategory) : Category == SetCategory
 with
   negative : V -> $
   null     : V -> $
   positive : V -> $
```

Les comparaisons à zéro d'expressions peuvent être simplifiées si l'expression comparée à zéro est de signe constant. Pour calculer ce signe, nous avons utilisé le package prédéfini `ElementaryFunctionSign`. Si le signe calculé est opposé au signe déclaré dans la comparaison, la condition est résolue à `False`. Le système fait alors les déclarations (par `assert`) qui permettent au système de vérifier la complétude des expressions conditionnelles. En effet, si $negative(a^2+b^2)$ est remplacé par `False`, $negative(a^2+b^2) \vee null(a^2+b^2) \vee positive(a^2+b^2)$ ne peut être calculé en vrais que si on sait que $null(a^2+b^2) \vee positive(a^2+b^2)$ est toujours vrais. Nous avons donc choisi de simplifier ces conditions modulo des équivalences[4] entre conditions telles que : $x(x+y) = 0 \Leftrightarrow x = 0 \vee x + y = 0$.

Les domaines `ExpressionCondition` (abbrégé en `EXC`) de comparaison à zéro des expressions sur `R` regardées comme des disjonctions de conjonctions de comparaisons à zéro d'expression sont donc définis par :

```
EXC R : Exports == Implementation where

 R : OrderedSet

 V    ==> Expression R
 E    ==> ECTZ V
 Equi ==> Record (lhs:$,rhs:$)

 Exports ==> Join (DCCAT E, EBMACAT Equi, CTZCAT V)
```

Remarque 5 *Lorsque l'expression à comparer à zéro est un quotient d/n, on compare à zéro $d * n$ puisque le signe de ces deux expressions est le même.*

4. On trouvera plus d'explication sur ces équivalences dans la description de l'implémentation.

Remarque 6 *Lorsque l'expression comparée à zéro est une fraction de polynômes dont le signe n'est pas constant, on met les conditions en forme normale en utilisant les formes primitives du numérateur et du dénominateur.*

2.4.2 Implémentation

Représentation

Nous avons implémentés les foncteurs de domaines `ElementaryComparisonsToZero` (abbrégé en `ECTZ`) appartenant à la catégorie `ElementaryComparisonsToZeroCategory` (abbrégé en `ECTZCAT`) et `DisjunctiveComparisonsToZero` (abbrégé en `DCTZ`) :

```
ECTZ V : Exports == Implementation where

 V : SetCategory

 Exports ==> ECTZCAT V

DCTZ (E,V):Exports == Implementation

 where

  V : SetCategory
  E : ECTZCAT V

  Equi ==> Record (lhs:$,rhs:$)

  Exports ==> Join (DCCAT E, EBMACAT Equi, CTZCAT V)
```

Les conditions de comparaison à zéro des expressions sur `R` notées `EXC` sont alors implémentées comme : `DCTZ (E,Expression R)` où `E` est implémenté comme `ECTZ Expression R`.

Les `CEX` sont représentées comme des :
`CO (EXC R,Expression R)`.

2.4.3 Application : l'intégration

Dans chaque système de Calcul Formel l'intégration est interprétée différemment, des recherches sont effectuées actuellement concernant la validité de ces résultats (voir [D.J93]). `Axiom`, rend comme résultat de l'intégration d'une expression les formes générales [5] de l'intégrale.

Exécution 1 *Quelques calculs d'intégrales :*

```
(1) -> integrate(1/x**2,x)

          1
   (1)  - -
          x
                       Type: Union(Expression Integer,...)
```

5. Il ne calcule pas les formes sur les valeurs critiques.

```
(2) ->integrate (1/(a*(x**2+2*x+1)),x)

            1
   (2)  - -------
          a x + a
                       Type: Union(Expression Integer,...)
```

```
(3) -> integrate (1/(x**2-a),x)

                                                      x
                                                 atan(------)
              2      +-+                 2            +---+
       log((x  + a)\|a  - 2a x) - log(x  - a)       \|- a
   (3)  [-------------------------------------,------------]
                          +-+                       +---+
                        2\|a                      \|- a
                 Type: Union(List Expression Integer,...)
```

```
(4) ->integrate(1/(x*(log(x)**2+a)),x)

   (4)
              2      +---+                          2
    log((log(x)  - a)\|- a  + 2a log(x)) - log(log(x)  + a)
   [-------------------------------------------------------,
                             +---+
                           2\|- a
        log(x)
    atan(------)
          +-+
         \|a
    ------------]
         +-+
        \|a
                 Type: Union(List Expression Integer,...)
```

Nous avons modifié l'intégrateur d'`Axiom` pour qu'il rende non pas la liste de toutes les valeurs possibles de l'intégrale, mais une expression conditionnelle qui exprime ces valeurs en fonction de conditions sur les paramètres, et n'oublie pas les valeurs critiques.

Pour construire l'intégrale conditionnelle, on pouvait soit faire un calcul conditionnel (avec un scindage à chaque choix), soit laisser le calcul tel qu'il était et modifier la construction du résultat après avoir défini les domaines de validité.
Nous avons choisi la deuxième solution, et étudié les domaines de validité des différentes valeurs de l'intégrale données par l'intégrateur d'`Axiom`.

Dans `Axiom`, il existe plusieurs fonctions d'intégration dévolues chacune à une classe de fonctions. Nous n'avons pour l'instant modifié qu'un seul de ces intégrateurs, les autres sont en cours de transformation.
On peut dire de façon générale que l'intégrale d'une expression de la forme

$$\int \frac{1}{ax^2+bx+c}$$

dépend du signe de

$$\delta = b^2 - 4ac.$$

Si $\delta = 0$, l'intégrale est polynomiale, si $\delta > 0$ elle est logarithmique, sinon elle est à base d'arc tangents.

La fonction d'intégration dévolue à cette classe de fonctions construit les valeurs de l'intégrale dans la fonction `quadratic` (du fichier `irexpand`).
Si δ possède un signe constant, `Axiom` rend l'unique valeur possible, sinon il rend une liste de deux intégrales I_1, I_2 où I_1 est la solution pour $\delta > 0$ et I_2 pour $\delta < 0$.

Dans le cas où le signe de δ change, il faut calculer les formes dégénérées (quand δ s'annule) de l'intégrale qui ne sont pas rendues par `Axiom`.
Si δ s'écrit $P(x)/Q(x)$ il nous faut savoir ce qui se passe quand P ou Q s'annule, c'est à dire quand $P(x)*Q(x)$ s'annule. Pour cela, nous avons résolu l'équation $P(x)Q(x) = 0$ par rapport à un paramètre x (en utilisant `solve`), puis obtenu la liste des zéros $[v_1, \cdots v_n]$ et calculé l'intégrale des différentes fonctions obtenues en évaluant $\frac{1}{ax^2+bx+c}$ en $x = v_1$,... $x = v_n$. En remarquant que le signe de $P(x)/Q(x)$ et de $P(x)Q(x)$ sont les mêmes, nous avons construit le résultat comme suit :

$$\begin{pmatrix} P(x)Q(x) > 0 & \rightarrow & I_1 \\ P(x)Q(x) < 0 & \rightarrow & I_2 \\ x = v_1 & \rightarrow & I_3 \\ \cdots & & \\ x = v_n & \rightarrow & I_{n+2}. \end{pmatrix}$$

Remarque 7 *Pour que le système puisse reconnaître que l'expression conditionnelle précédente est complète, la déclaration*

$$P(x)Q(x) = 0 \Leftrightarrow \vee_i (x = v_i)$$

est faite par la fonction `quadratic` *(on peut en voir un exemple*`(29)(30)(31)` *dans la section suivante).*

2.5 Quelques calculs

2.5.1 Calcul de conditions sur les expressions

```
Starts dribbling to CEX.out (1993/12/21, 8:34:25).
(1) ->)read CEX
)lo EBCAT EBCAT- EBMACAT EBMACAT- ECCAT DCCAT CTZCAT ECTZCAT
)lo DC DCME ECTZ DCTZ
)lo PFS EXC

   loading ...

assert (null (a*(b-1)),Or (null a,null (b-1)))$EXC INT

   (1)  [lhs= (a b - a=0),rhs=  Or   (a=0)  ]
                                    (b - 1=0)
Type: Record(lhs: ExpressionCondition Integer,rhs: ExpressionCondition Integer)
simplify (And (Implies(null (a*(b-1)),Or (null a,null (b-1))),_
              Implies (Or (null a,null (b-1)),null (a*(b-1)))))$EXC INT

   (2)  True
                                              Type: ExpressionCondition Integer
simplify (Implies (Or (null a,null (b-1)),null (a*(b-1))))$EXC INT

   (3)  True
                                              Type: ExpressionCondition Integer
```

```
simplify (Implies (null (a*(b-1)),Or (null a,null (b-1))))$EXC INT

   (4)  True
                                                Type: ExpressionCondition Integer
simplify (And (positive (a*(b-1)),null a))$EXC INT

   (5)  False
                                                Type: ExpressionCondition Integer
simplify (And (null (a*(b-1)),null (b-1)))$EXC INT

   (6)  (b - 1=0) And (a b - a=0)
                                                Type: ExpressionCondition Integer
assert (null (a**2+b**2),And (null a,null b))$EXC INT

                2    2
   (7)  [lhs= (b  + a =0),rhs= (a=0) And (b=0)]
Type: Record(lhs: ExpressionCondition Integer,rhs: ExpressionCondition Integer)
simplify (Implies (And (null a,null b),null (a**2+b**2)))$EXC INT

   (8)  True
                                                Type: ExpressionCondition Integer
simplify(Implies (null (a**2+b**2),And (null a,null b)))$EXC INT

   (9)  True
                                                Type: ExpressionCondition Integer
simplify (Or [positive (a**2+b**2),negative (a**2+b**2),_
              And (null a,null b)]$EXC INT)

   (10)  True
                                                Type: ExpressionCondition Integer
simplify (And (null (a**2+b**2),null a))$EXC INT

                     2    2
   (11)  (a=0) And (b  + a =0)
                                                Type: ExpressionCondition Integer
simplify (And (null (a**2+b**2),positive a))$EXC INT

   (12)  False
                                                Type: ExpressionCondition Integer
assert (positive (a**2+b**2)$EXC INT,Not And (null a,null b)$EXC INT)

                2    2
   (13)  [lhs= (b  + a >0),rhs=  Or (a<0)]
                                    (a>0)
                                    (b<0)
                                    (b>0)
Type: Record(lhs: ExpressionCondition Integer,rhs: ExpressionCondition Integer)
simplify (Implies (Not And (null a,null b)$EXC INT,_
                   positive (a**2+b**2)$EXC INT))

   (14)  True
                                                Type: ExpressionCondition Integer
simplify (Implies (positive (a**2+b**2)$EXC INT,_
          Not And (null a,null b)$EXC INT))

   (15)  True
                                                Type: ExpressionCondition Integer
simplify (Or (positive (a**2+b**2),null (a**2+b**2)))$EXC INT
```

```
   (16)  True
                                                 Type: ExpressionCondition Integer
```

2.5.2 Calcul d'expressions conditionnelles

```
)lo PCCAT PC CCAT CO PCF       -- conditional
)lo CEX CPICPAC                -- conditionalExpression

   loading ...

(abs (abs (n*(n+1)/2) + abs n))$CEX INT

                                                                  2
                          2              2              2   - n  + n
   (17)   ( Or (n<0) And (n  - n<0) And (n  + n>0) -> --------)
                          2              2              2
               (n>0) And (n  - n<0) And (n  + n<0)
                             2              2
             ( Or (n<0) And (n  - n=0) And (n  + n>0)  -> 0)
                             2              2
                 (n<0) And (n  + n<0) And (n  + 3n=0)
                                   2
                          (n=0) And (n  + n=0)
                             2              2
                 (n>0) And (n  - n=0) And (n  + n<0)
                             2              2
                 (n>0) And (n  + n>0) And (n  + 3n=0)
                                                          2
                           2              2           n  - n
           ( Or (n<0) And (n  - n>0) And (n  + n>0) -> ------)
                           2              2              2
                (n>0) And (n  - n>0) And (n  + n<0)
                                                       2
                          2              2           - n  - 3n
         ( Or (n<0) And (n  + n<0) And (n  + 3n<0) -> ---------)
                          2              2                2
              (n>0) And (n  + n>0) And (n  + 3n<0)
                                                      2
                          2              2           n  + 3n
          ( Or (n<0) And (n  + n<0) And (n  + 3n>0) -> -------)
                          2              2                2
               (n>0) And (n  + n>0) And (n  + 3n>0)
                                  2
                       ((n<0) And (n  + n=0) -> - n)
                                                 2
                                2           - n  - n
                     ((n=0) And (n  + n<0) -> --------)
                                                 2
                                               2
                                2           n  + n
                      ((n=0) And (n  + n>0) -> ------)
                                                 2
                                  2
                         ((n>0) And (n  + n=0) -> n)
                                                 Type: ConditionalExpression Integer
```

```
(a+b)$CEX INT

   (18)  (True -> b + a)
                                          Type: ConditionalExpression Integer
(a*b)$CEX INT

   (19)  (True -> a b)
                                          Type: ConditionalExpression Integer
(abs x)$CEX INT

   (20)  ((x<0) -> - x)
          ((x=0) -> 0)
          ((x>0) -> x)
                                          Type: ConditionalExpression Integer
(abs abs x)$CEX INT

   (21)  ((x<0) -> - x)
          ((x=0) -> 0)
          ((x>0) -> x)
                                          Type: ConditionalExpression Integer
(abs (x + abs y))$CEX INT

   (22)      ((x<0) And (y=0) -> - x)
          ( Or   (x=0) And (y=0)   -> 0)
               (y - x=0) And (y<0)
               (y>0) And (y + x=0)
              ((x>0) And (y=0) -> x)
         ((y - x<0) And (y<0) -> - y + x)
          ((y - x>0) And (y<0) -> y - x)
         ((y>0) And (y + x<0) -> - y - x)
          ((y>0) And (y + x>0) -> y + x)
                                          Type: ConditionalExpression Integer
(abs (x) * abs (x+y))$CEX INT

                                            2
   (23)   ( Or (x<0) And (y + x<0) -> x y + x )
               (x>0) And (y + x>0)
             ( Or (x<0) And (y + x=0) -> 0)
                  (x=0) And (y + x<0)
                  (x=0) And (y + x=0)
                  (x=0) And (y + x>0)
                  (x>0) And (y + x=0)
                                           2
          ( Or (x<0) And (y + x>0) -> - x y - x )
               (x>0) And (y + x<0)
                                          Type: ConditionalExpression Integer
(abs a**2)$CEX INT

                       2
   (24)  ( Or (a<0) -> a )
              (a>0)
           ((a=0) -> 0)
                                          Type: ConditionalExpression Integer
(gcd (a,b))$CEX INT

   (25)  (True -> 1)
                                          Type: ConditionalExpression Integer
(quo (a,b))$CEX INT
```

```
                a
   (26)  (True -> -)
                b
                                        Type: ConditionalExpression Integer
((a+5)/b)$CEX INT

                a + 5
   (27)  (True -> -----)
                  b
                                        Type: ConditionalExpression Integer
operators %

   (28)  [a,b]
                                                    Type: List BasicOperator
```

2.5.3 Calcul d'intégrales

```
)lo CFSINT IR2CF

   loading ...

clearAssertions()$EXC INT

   (29)   []
Type: List Record(lhs: ExpressionCondition Integer,rhs: ExpressionCondition Integer)
integrate (a/(x**2-a**2+1) :: EXPR INT,x)

                                                  1
   (30)                                  ((a - 1=0) -> - -)
                                                  x
                                            ((a=0) -> 0)
                                                  1
                                           ((a + 1=0) -> -)
                                                  x
                                                      x
                                             a atan(-----------)
                                                     +--------+
                                                     |   2
                                    4    2           \|- a  + 1
                                 ((a  - a <0) -> -------------------)
                                                   +--------+
                                                   |   2
                                                  \|- a  + 1
                                              +------+
                                   2    2     | 2            2                 2    2
          4    2       a log((x  + a  - 1)\|a  - 1  + (- 2a  + 2)x) - a log(x  - a  + 1)
       ((a  - a >0) -> ------------------------------------------------------------------)
                                                   +------+
                                                   | 2
                                                  2\|a  - 1
                                        Type: ConditionalExpression Integer
assertions()$EXC INT

                4    2
   (31)  [[lhs= (a  - a =0),rhs=  Or (a - 1=0)]]
```

```
                                    (a=0)
                                  (a + 1=0)
Type: List Record(lhs: ExpressionCondition Integer,rhs: ExpressionCondition Integer)
integrate ((1/(x**2-a))::EXPR INT,x)

                                     x
                                atan(------)
                                     +---+
                                     \|- a
   (32)               ((a<0) -> ------------)
                                   +---+
                                   \|- a
                                     1
                          ((a=0) -> - -)
                                     x
                                2      +-+                2
                     log((x  + a)\|a  - 2a x) - log(x  - a)
           ((a>0) -> ------------------------------------)
                                   +-+
                                   2\|a
                                          Type: ConditionalExpression Integer
integrate ((1/(x**2- (a**2+b**2)))::EXPR INT,x)

                                   +-------+
                        2    2    2  | 2    2         2     2           2    2    2
                   log((x  + b  + a )\|b  + a   + (- 2b  - 2a )x) - log(x  - b  - a )
   (33)  (True -> ------------------------------------------------------------------)
                                        +-------+
                                        | 2    2
                                      2\|b  + a
                                          Type: ConditionalExpression Integer
integrate ((1/(a*(x**2+2*x+1)))::EXPR INT,x)

                        1
   (34)  (True -> - -------)
                   a x + a
                                          Type: ConditionalExpression Integer
integrate (1/(x*(log(x)**2+a)),x)

                              2      +---+                          2
                   log((log(x)  - a)\|- a  + 2a log(x)) - log(log(x)  + a)
   (35)  ((a<0) -> ------------------------------------------------------)
                                        +---+
                                      2\|- a
                                         1
                               ((a=0) -> - ------)
                                          log(x)
                                           log(x)
                                     atan(------)
                                            +-+
                                            \|a
                                ((a>0) -> ------------)
                                            +-+
                                            \|a
                                          Type: ConditionalExpression Integer
integrate ((1/(x * (a*x**2 + b*x + c)))::EXPR INT,x)

   (37)
```

```
      2    2        (- b x - 2c)log(b x + 2c) + (b x + 2c)log(x) + 2c
((4a c  - b c=0) -> ------------------------------------------------)
                                          2
                                   b c x + 2c

      2 3    4 2
((4a b c  - b c <0) ->

           b
        *
           log
                       2 2        2      4  2             2     3           3     2 2
                  ((18a c  - 20a b c + 4b )x  + (- 30a b c  + 8b c)x - 18a c  + 5b c )
               *
                   +-----------+
                   |          2
                  \|- 4a c + b
              +
                        2   2        3      5  2         2 3        2 2     4             3
                (- 48a b c  + 28a b c - 4b )x  + (- 72a c  + 50a b c  - 8b c)x + 12a b c
              +
                     3 2
                - 3b c
      +
                                          +-----------+
                   2                      |          2             2
        (- log(a x  + b x + c) + 2log(x))\|- 4a c + b   - b log(a x  + b x + c)
   /
         +-----------+
         |          2
        2c\|- 4a c + b

      2 3    4 2
((4a b c  - b c >0) ->

                   +---------+
                   |        2
                 b\|4a c - b              - 2a x - b
        2b atan(-------------) + 2b atan(------------)
                          2               +---------+
                   3a c - b               |        2
                                         \|4a c - b
      +
                                          +---------+
                   2                      |        2
        (- log(a x  + b x + c) + 2log(x))\|4a c - b
   /
         +---------+
         |        2
        2c\|4a c - b
                                    Type: ConditionalExpression Integer
```

Chapitre 3

Objets Inconnus

Dans ce chapitre, nous allons définir la notion d'inconnue, spécifier les domaines d'objets inconnus et donner un exemple d'inconnues les inconnues entières.

3.1 Définition

- Qu'est-ce qu'un objet inconnu ?
  Un objet inconnu est soit un symbole typé (inconnue de base), soit une expression construite à partir d'inconnues de bases, de valeurs appartenant à des ensembles compatibles et des opérateurs valides.
  La règle de construction de ces objets peut être énoncée de la manière suivante : **une inconnue est valide si lorsqu'on remplace toutes les inconnues de base qu'elle contient par des valeurs et les opérateurs par des fonctions, elle est évaluée en une "bonne" valeur.**

- Que dire des inconnues de base ?
  Il faut les choisir avant la construction des expressions, c'est à dire leur donner un nom (un symbole) ainsi qu'un "type".
  Les opérateurs définis sur une classe d'inconnues doivent pouvoir agir sur tous les objets de cette classe, donc en particulier sur toutes ces inconnues de base.

- Que dire des valeurs ?
  Comme précédemment il faut que les opérateurs définis sur une classe d'inconnues puissent agir sur toutes les valeurs.

- Que dire des valeurs et des inconnues de base ?
  La règle de construction des expression montre un lien fort entre le type des inconnues de base et l'ensemble des valeurs, puisqu'un ensemble d'inconnues doit être stable par remplacement des inconnues de bases par des valeurs : le type des inconnues de base et l'ensemble des valeurs sont compatibles.

Les différents points décrits précédemment peuvent être mis en pratique de façons différentes, nous n'irons pas plus loin dans leur description à ce niveau général.

3.2 Spécification

Deux points de vue différents concernant les objets inconnus peuvent exister suivant la philosophie du système de Calcul Formel dans lequel il est implémenté.
On peut soit considérer une classe générale d'expression appelée classe des objets inconnus, soit découper cette classe générale en classes typées (les entiers inconnus, les polynômes inconnus ...).

Le premier point de vue a été mis en application dans le système prototypal `ulysse` (voir [Fau92]). Nous allons maintenant présenter le deuxième point de vue et construire donc un domaine d'inconnues par "type formel".

3.2.1 Domaine d'inconnues

Nous avons choisi de considérer que toutes les inconnues de base d'un même domaine d'inconnues ont même "type", et que les valeurs appartenant à ce domaine ont aussi même "type" car il semble impossible de définir des opérateurs internes à ce domaine sans cela.
On peut remarquer que le type des inconnues de base est un type "formel" (ensemble mathématique), c'est à dire qu'il n'est pas lié à une représentation alors que le type des valeurs est un type de données.

Un constructeur de domaines d'inconnues est donc défini à partir de trois informations :

1. une liste de symboles représentants les noms des inconnues de base (notée `BasicUnknown`),
2. un type formel (noté `X`),
3. un domaine de valeurs (noté `D`).

Si on définit un unique foncteur de domaines d'inconnues, celui-ci possède un type très compliqué. Tout d'abord, les opérateurs définis pour un domaine d'inconnues doivent être valides pour le domaine de valeurs (`D`), ainsi que pour les inconnues de base (`BasicUnknown`). Le type du domaine d'inconnues est donc le type de `D`.
En fait ce n'est pas vraiment le type au sens d'`Axiom` qu'on veut conserver, mais le type mathématique. En effet, les opérations purement informatique prises en compte dans les types d'`Axiom` telles que `shift` sur les entiers n'ont plus de sens sur les inconnues entières.

Nous avons choisi de définir un constructeur par type mathématique, c'est à dire ensemble d'opérations munies de propriétés.
Les constructeurs de domaines d'inconnues les plus généraux semblent donc posséder une en-tête de la forme :

```
UnknownX (D:X,BasicUnknown:List Symbol):X
```

Bien sûr, le reste de la définition du domaine dépendra des propriétés de la catégorie `X`.

3.2.2 Inconnues et autres types d'expressions

Les inconnues peuvent être "mélangées" à d'autres types d'expressions, il faut donc définir des conversions permettant à `Axiom` de déterminer le type de ces expressions mixtes.
De manière standard, le système cherche à transformer toute expression mixte en une inconnue du type de l'inconnue qu'elle contient.

Nous avons décidé d'assimiler les variables et les inconnues de base de même nom présentent dans une expressions, car nous considérons qu'un symbole ne peut avoir qu'une unique sémantique dans une expression. Si une expression mixte contient des inconnues de base, des variables assimilables aux inconnues de base et des opérateurs valides sur ces inconnues, le système doit donc construire une inconnue.

Par contre, que doit-t'il faire si certaines variables ne sont pas assimilables aux inconnues de base ?
Trois comportements sont possibles[1] :

1. le système provoque une erreur (c'est le comportement actuel) :

```
(1) -> m::UR ([n,m,p],Integer) + x

   >> Error detected within library code:
   it's not an UnknownInteger
   You are being returned to the top level of the
   interpreter.
```

2. le système déstructure l'inconnue :

```
(1) -> m:UR ([n,m,p],Integer) + x

   (1)  x + m
                                    Type: Polynomial Integer
```

   Ceci est le comportement adopté pour le type prédéfini `MultivariatePolynomial` qui ressemble structurellement au type des inconnues, puisqu'il est paramétré par une liste de variables et un domaine de coefficients.

3. le système utilise le type `UR` comme type des coefficients :

```
(1) -> m:UR ([n,m,p],Integer) + x

   (1)  x + m
                  Type: Polynomial UR ([n,m,p],Integer)
```

Exécution 2 *Voici quelques exemples de ce que nous voulons obtenir ...*

```
(1) -> a : UR ([n,m,p],Integer)
                                Type: Void
(2) -> a := n+m

   (2)  n + m
                                Type: UR([n,m,p],Integer)
(3) -> a + n

   (3)  2n + m
                                Type: UR([n,m,p],Integer)
(4) -> b : UR ([n],Integer)
                                                Type: Void
(5) -> b := n + 1

   (5)  n + 1
                                    Type: UR([n],Integer)
(6) -> a + (b::UR([n,m,p],Integer))

   (6) 2n + m + 1
                                Type: UR([n,m,p],Integer)
```

1. On note `UR` (UnknownRing) le type des inconnues qui forment un anneau.

Nous avons dégagé les règles générale régissant le choix du type vers lequel on peut convertir deux expressions de types $Type_1$ et $Type_2$:

$Type_1$	$Type_2$	*Type resultat*
$U(B, D_1)$	D_2	$U(B, D_1)$ *si* $D_2 \rightarrow D_1$ *error*
$U(B, D)$	$s : Symbol$	$U(B, D)$ *si* $s \in B$ *error*
$U(B1, D1)$	$U(B2, D1)$	$U(B1, D1)$ *si* $B2 \in B1$ *error*
$U(B1, D1)$	$U(B1, D2)$	$U(B1, D1)$ *si* $D2 \rightarrow D1$ *error*

Les règles précédentes décrivent les règles générales de conversion. Mais suivant le type de $U(B, D)$ on aura envie d'ajouter d'autres conversions. Par exemple dans le cas ou $U(B, D)$ est un anneau (D est aussi un anneau), on voudra faciliter le travail de l'utilisateur en créant des conversions entre `UR(B,D)` et `Polynomial D`.

Exécution 3 *Voici la suite de l'exécution précédente présentant les liens particuliers entre* `UR(B,D)` *et* `Polynomial D` *:*

```
(7) -> a + x

   (7) a + x
                    Type: Polynomial UR([n,m,p],Integer)
(8) -> (n+m)::UR([n,m,p],Integer)

   (8) n + m
                               Type: UR([n,m,p],Integer)
```

D'où les règles supplémentaires:

$Type_1$	$Type_2$	*Type resultat*
$UR(B, D_1)$	D_2	$Polynomial\ D_2$ *si* $D_1 \rightarrow D_2$
$UR(B, D)$	$s : Symbol$	$Polynomial\ UR(B, D)$ *si* $s \notin B$

3.2.3 Catégorie des domaines d'inconnues

Dans la section précédente, nous avons défini les fonctions de conversions entre les inconnues et les autres types d'expressions nécessaires à la bonne utilisation de ces inconnues.

Nous avons pris en compte ces conversions pour définir la catégorie générale de domaines d'inconnues `UnknownCategory` (abbrégé en `UCAT`) :

```
UCAT (BasicUnknown,D):Category
   == Exports where
```

```
 D:SetCategory
 BasicUnknown:List Symbol

 Exports ==> SetCategory with
    RetractableTo OrderedVariableList (BasicUnknown)
    RetractableTo Symbol
```

et celles des domaines d'inconnues qui sont des anneaux `UnknownRingCategory` (abbrégé en `URCAT`) :

```
URCAT (BasicUnknown,D):Category
   == Exports where

 D:Ring
 BasicUnknown:List Symbol

 Exports ==> Join (UCAT (BasicUnknown,D),Ring) with
    retractIfCan : Polynomial D -> Union ($,"failed")
```

Les autres conversions doivent être définies dans un `package` car elles mettent en relation deux domaines différents d'inconnues.
L'en-tête de ces package est de la forme :

```
ConversionsPackage (B1,B2,D1,D2):Exports == Implementation
 where
  D1,D2 : SetCategory
  B1,B2 : List Symbol

  U1 : UCAT (B1,D1)
  U2 : UCAT (B2,D1)
  U3 : UCAT (B1,D2)

  Exports ==> with
   coerce : U1 -> U2
   coerce : U1 -> U3.
```

3.3 Exemple : les entiers

3.3.1 Définition

Les opérations définies sur les inconnues entières pures sont des opérations d'anneau, la plus petite classe d'expressions qui peut les contenir toutes est celle des polynômes sur les rationnels.
Il est clair que tous les polynômes à coefficients entiers sont des inconnues entières car la somme, le produit de deux entiers et la puissance entière d'un entier sont des entiers.
Nous allons maintenant définir le sous ensemble des polynômes à coefficients rationnels qui sont des inconnues entières pures ($n^2/2 + n/2$).

Proposition :
Les seuls polynômes de $Q[x_1, ...x_n]$ qui sont des inconnues entières peuvent se réécrire sous la forme :

$$p(x_1, ..., x_n) = \sum_j a_j(x_2, ..., x_n) * basic_j(x_1)$$
$$basic_j(x_i) = \prod_{k=0}^{j-1}(x_i + k)/j!$$

où $\forall\ j,\ a_j(x_2, ..., x_n)$ est aussi un entier.

Preuve (en une variable) :
Soit $f(x) \in Q[x]$ telle que $f : Z \rightarrow Z$ et $n = degre(f)$, on peut toujours écrire f comme :

$$\sum_{i=0}^{n} C_i \left(\begin{array}{c} x \\ i \end{array} \right)$$

où $\forall i, C_i \in Q$.
Il nous reste à démontrer que :

$$\forall i, C_i \in Z.$$

Lemme :

$$\begin{array}{rcl} \sum_{a=b}^{n} \left(\begin{array}{c} n \\ a \end{array} \right) \left(\begin{array}{c} a \\ b \end{array} \right) (-1)^{n-a} & = & 0 \quad si \quad b < n \\ & = & 1 \quad si \quad b = n \end{array}$$

Preuve du Lemme :

$$\begin{array}{rcl} 1 & = & (1 + x - x)^n \\ & = & \sum_{a=0}^{n} \sum_{b=0}^{a} x^{n-b} (-1)^{n-a} \left(\begin{array}{c} n \\ a \end{array} \right) \left(\begin{array}{c} a \\ b \end{array} \right) \\ & = & \sum_{b=0}^{n} x^{n-b} \sum_{a=b}^{n} (-1)^{n-a} \left(\begin{array}{c} n \\ a \end{array} \right) \left(\begin{array}{c} a \\ b \end{array} \right) \end{array}$$

A gauche de cette égalité nous avons un polynôme en x, qui doit être égal à 1, donc tous ses coefficients sont nuls sauf pour $n - b = 0$.

Preuve de la proposition :
Soit à calculer l'expression :

$$\sum_{x=0}^{m} f(x)(-1)^{m-x} \left(\begin{array}{c} m \\ x \end{array} \right)$$

$$\sum_{x=0}^{m} f(x)(-1)^{m-x} \left(\begin{array}{c} m \\ x \end{array} \right) = \sum_{i=0}^{n} C_i \sum_{x=0}^{m} (-1)^{m-x} \left(\begin{array}{c} m \\ x \end{array} \right) \left(\begin{array}{c} x \\ i \end{array} \right)$$

Comme

$$x < i \Rightarrow \left(\begin{array}{c} x \\ i \end{array} \right) = 0,$$

donc

$$\begin{array}{rcl} \sum_{x=0}^{m} f(x)(-1)^{m-x} \left(\begin{array}{c} m \\ x \end{array} \right) & = & \sum_{i=0}^{n} C_i \sum_{x=i}^{m} (-1)^{m-x} \left(\begin{array}{c} m \\ x \end{array} \right) \left(\begin{array}{c} x \\ i \end{array} \right) \\ & = & C_m. \end{array}$$

Conclusion :

$$f(x) \in Z \Rightarrow \sum_{x=0}^{m} f(x)(-1)^{m-x} \left(\begin{array}{c} m \\ x \end{array} \right) \in Z \Rightarrow \forall\ m \in [0..n] \quad C_m \in Z.$$

Cette proposition définit un critère de choix des polynômes à coefficients rationnels qui sont des inconnues entières.

3.3.2 Spécification

La catégorie `IntegerNumberSystem` contient toutes les implémentations des entiers, les paramètres du foncteur de domaines des inconnues entières sont donc typés comme suit: `D:IntegerNumberSystem` et `BasicUnknown:List Symbol`.

Les opérations définies sur les entiers que nous voulons étendre aux inconnues entières sont de trois sortes:

1. les opérations d'anneau,
2. l'opérateur "/",
3. les opérateurs "$abs, <, =$".

Les opérations d'anneau ainsi que l'opérateur "/" rendent des objets mono-valués alors que les trois opérateurs suivants ne sont intéressants que si ils rendent des objets conditionnels. Par exemple, le calcul de $2 * abs(n) - abs(2 * n)$ en 0 ne peut se faire que si le système connaît `abs`.

Nous appelerons **inconnues pures**, les inconnues constituées uniquement des opérateurs `+,*,...,/`, alors que celles qui contiennent l'opérateur `abs` sont des inconnues conditionnelles. Les opérateurs `<,=` sont des opérateurs externes qui construisent des booléens conditionnels.

Inconnues entières pures

A partir de symboles, d'entiers et des opérateurs `+,*,...,/`, le système `Axiom` construit une expression de type `Polynomial Fraction D`. Pour faciliter la construction des inconnues entières pures `PureUnknownInteger` (abbrégé en `PUI`), nous avons ajouté aux conversions de `URCAT` les conversions entre `PUI` et `Polynomial Fraction D`.

De plus, on veut pouvoir utiliser la structure particulière des inconnues entières pures pour les construire et les décomposer aisément, nous avons donc exporté des opérateurs pour faire ce travail: `destruct` qui décompose une inconnue entière pure par rapport aux $basic_i(s)$ (voir définition page 31) où `s:Symbol`, alors que `construct` fait le travail inverse.

Le constructeur de domaines de `PUI` est donc défini comme:

```
PureUnknownInteger (D,BasicUnknown):Exports == Implementation
  where
 D:IntegerNumberSystem,
 BasicUnknown:List Symbol

 Exports ==> URCAT with
   "/": ($,D) -> $
   "/" : ($,$) -> $
   destruct : ($,Symbol) -> List $
   construct : (List $,Symbol) -> $
   retractIfCan : Polynomial Fraction D -> Union ($,"failed")
```

Conditions sur les inconnues entières pures

Les conditions nécessaires à la définition des opérateurs `abs,<,=` sont des disjonctions de conjonctions de comparaisons à zéro d'inconnues entières pures.

Dans le cas des comparaisons à zéro d'entiers, on peut mettre en oeuvre plus de simplifications que précédemment car on peut plus facilement connaître le signe d'un polynôme que d'une expression. Nous avons introduit deux classes de simplifications: une première qui permet effectivement de réduire la taille des expressions et est utilisée automatiquement, et l'autre qui peut réduire la taille des expressions dans certains cas et qui est mise à la disposition des utilisateurs sous forme de fonctions.

Tout d'abord, on peut remarquer que toute inconnue entière pure peut s'écrire comme un quotient P/n où P est un polynôme sur D et n appartient à D. De plus, le signe de P/n est égal au produit des signes de $X = unit(P) * n * content(P)$ et de $Y = (P * unit(P))/content(P)$. Comme dans notre cas X est un nombre entier (élément de D), il possède un signe constant. Nous avons donc choisi une forme normale pour les conditions calculée par application de la règle:

$$sign(P/n) \rightarrow X * sign(Y).$$

Ceci permet donc de reconnaître que les conditions élémentaires $positive(-3(x+y))$ et $negative(x+y)$ sont équivalentes. Et enfin, dans certains cas on peut calculer le signe effectif de Y ($a^2 + b^2$ est positif ou nul), il faut alors que ce signe soit en accord avec le signe demandé, sinon la condition est réduite à `False` (par exemple $negative(x^2 + y^2)$ sera remplacé par `False`).
De même, il est possible de déterminer des contradictions et des redondances entre deux conditions élémentaires $(sign_1(x), sign_2(y))$ par l'étude du signe de la différence $x - y$.
En effet si $x - y$ est positif, x est toujours supérieur à y. Si les signes demandés $sign_1$ et $sign_2$ ne respectent pas cette contrainte, la conjonction $sign_1(x) \wedge sign_2(y)$ est réduite à `False`. Si par contre ils respectent cette contrainte et que les signes demandés définissent des intervalles qui se recouvrent ($x > y$ et $x > 0 \wedge y > 0$, alors la conjonction est réduite à une condition élémentaire $y > 0$. Ces simplifications sont utilisées automatiquement par le système.

Enfin, on peut utiliser plus complètement ces simplifications automatiques en décomposant les conditions élémentaires suivant les polynômes non factorisables. En effet, la conditions élémentaire $sign \prod_i p_i$ se décompose aisément en fonction des signes des p_i. Or le signe effectif d'un produit peut être inconnu alors que les signes de certains de ses facteurs sont eux connus, donc des simplifications peuvent apparaître dans la nouvelle forme qui ne pouvaient être effectuée si la forme initiale. Nous avons appelé la fonction qui effectue ce calcul `expand` car elle développe les signes de produits.

Inconnues entières

Comme nous l'avons vu précédemment, certains opérateurs sur les inconnues entières calculent des inconnues entières conditionnelles, alors que d'autres construisent des booléens conditionnels.
Pour pouvoir utiliser l'opérateur `expand` défini sur les `PUICondition`, nous avons ajouté aux définitions de ces domaines conditionnels l'opérateur `expand` qui applique `expand` sur les conditions et construit l'objet conditionnel cohérent et complet équivalent à celui d'entrée.

Remarque 8 *Nous avons défini un domaine équivalent au domaine* `Boolean` *appelé* `CopyBoolean` *(abbrégé en* `CB`*) appartenant à* `EBCAT` *car on ne peut changer le type de* `Boolean`.

Nous avons donc défini un foncteur de domaines de booléens conditionnels dont les conditions sont des `PUIC` nommé `CUICopyBoolean` (abbrégé en `CUICB`) qui prend en paramètres la liste des inconnues de base ainsi que le domaine de base. Ces deux foncteurs sont définis ainsi:

```
CUICB (BasicUnknown,D):Exports == Implementation where

 D : IntegerNumberSystem
```

```
BasicUnknown : List Symbol

C   ==> PUIC (BasicUnknown,D)
V   ==> CB

Exports ==> Join (CCAT (C,V), EBCAT) with
  expand : $ -> $
```

Les entiers conditionnels `ConditionalUnknownInteger` (abbrégé en `CUI`) définis à partir de `BasicUnknown,D` forment un anneau, et exportent en plus `/,abs,inf?,eq?`[2], ils ont donc été définis comme:

```
CUI (BasicUnknown,D):Exports == Implementation where

  D : IntegerNumberSystem
  BasicUnknown : List Symbol

  OV   ==> OrderedVariableList BasicUnknown
  C    ==> PUIC (BasicUnknown,D)
  V1   ==> PUI (BasicUnknown,D)
  PCV1 ==> PC (C,V1)
  V2   ==> CB
  PCV2 ==> PC (C,V2)
  CV2  ==> ConditionalCopyBoolean (\c CV2)
  PD   ==> Polynomial D
  PFD  ==> Polynomial Fraction D
  PCF1 ==> PartialConditionalFunctions (C,V1,V1)
  PCF2 ==> PartialConditionalFunctions (C,V1,V2)

  Exports ==> Join (CCAT(C,V1), URCAT(OV,D)) with
    "/"    : ($,D) -> $
    "/"    : ($,$) -> $
    abs    : $ -> $
    expand : $ -> $
    inf?   : ($,$) -> CV2
    eq?    : ($,$) -> CV2
```

3.3.3 Implémentation

Conversions

Le problème principal que nous rencontré eu lors de l'implémentation des inconnues est celui des conversions. En effet, rendre effectives les règles que nous avons définies (voir 3.2.2) nous a demandé beaucoup d'efforts ...

Pour que les fonctions de conversions soient utilisées correctement, il faut de plus positionner le type des inconnues dans la variable `$multivariateDomains` par la déclaration: `)bo PUSH ('Unknown,$multivariateDomains)`.

Malgrès cela, il reste encore quelques problèmes dans ces conversions!

Exécution 4

2. Les noms d'opérateurs `<,=` ne peuvent être utilisés ici car il y aurait confusion entre les fonctions de types: `($,$) -> Boolean` et `($,$) -> CE(C,V)`.

```
(1) ->)bo PUSH ('PureUnknownInteger,$multivariateDomains)
(1) ->)bo PUSH ('ConditionalUnknownInteger,$multivariateDomains)
(1) -> n::PUI ([n,m,p],Integer)

   Cannot convert from type Variable n to PUI([n,m,p],Integer)
   for value n
(1) -> (n+m)::PUI ([n,m,p],Integer)

   (1)  n + m
                                      Type: PUI([n,m,p],Integer)
(2) -> (m * (m+1)/2)::PUI ([n,m,p],Integer)

         2
        m  + m
   (2)  ------
          2
                                      Type: PUI([n,m,p],Integer)
(3) -> a : PUI ([n,m,p],Integer)
                                                 Type: Void
(4) -> a := n + m

   (4)  n + m
                                Type: PUI([n,m,p],Integer)
(5) -> a + n + m

   (5)  2n + 2m
                                Type: PUI([n,m,p],Integer)
(6) -> a + x

   Cannot find a definition or library operation named + with
   argument types
                    PUI([n,m,p],Integer)
                    Variable x
```

Remarque 9 *Les fonctions* `retractIfCan` *sont en général exportées dans leur domaine de départ. Dans notre cas, elles sont exportées par celui d'arrivé, comme ce cas n'est pas prévu dans l'algorithme de recherche de ces fonctions, il nous a fallu modifier plus profondément le système. Nous avons modifié les fonctions* `coerceRetract` *et* `retractByFunction` *du fichier* `i-coerce`.

De plus, la variable globale `$multivariateDomains` *n'est pas définie dans la version actuelle d'*`Axiom`*, nous avons donc du modifier les fonctions* `isPolynomialMode` *de* `i-spec1.boot` *et* `polyVarlist` *de* `i-analy`. *Le chargement de ces fonctions, et la modification de la variable* `$multivariateDomains`*, se font par :*

```
(1) ->)fc polyVarlist )from i-analy
(1) ->)li (load "i-spec1-patch")
(1) ->)bo PUSH ('UnknownInteger,$multivariateDomains)
```

Remarque 10 *Lors de l'introduction d'un nouveau type dans* `Axiom`*, il n'existe qu'un nombre restreint de choix possible concernant les conversions entre les objets de ce type et les autres objets du système.*
Nous pensons que des modèles de packages de conversions pourraient être définis dans `Axiom` *pour réaliser les différents comportements possibles.*
Ceci aiderait grandement l'utilisateur !!!!

Calcul de signe effectif

Nous avons vu précédemment que pour simplifier les conditions de comparaison à zéro des inconnues entières, il était nécessaire de pouvoir calculer le signe d'un polynôme sur `D`.
Nous avons défini un package de nom `PolynomialFunctionSign` (abbrégé en `PFS`) qui exporte la fonction `Sign` qui calcule le signe d'un `Polynomial D`. Ce package est implémenté de façon tout à fait similaire au package `RationalFunctionSign` déjà défini dans `Axiom`.

```
PFS D : Exports == Implementation where

 D : IntegerNumberSystem

 PolD ==> Polynomial D
 SGN  ==> ToolsForSign D

 Exports ==> with
   sign : PolD -> Union (Integer,"failed")
```

Représentations

Nous avons choisi de représenter les inconnues pures comme des fractions dont le numérateur est un polynôme dont les variables sont les inconnues de base contenues dans `BasicUnknown`, les coefficients sont des entiers de `D` et les exposants sont des entiers du domaine `Integer` et le dénominateur un entier de `D`. Ce type est défini en `Axiom` par: `LocalAlgebra (Polynomial D,D,D)`.

La forme normale que nous avons choisie pour les conditions élémentaires `ElementaryPUICondition` (abbrégé en `EPUIC`), permet d'implémenter le domaine des conditions sur les inconnues entières pures nommé `PUICondition` (abbrégé en `PUIC`) comme un domaine de comparaisons à zéro de polynômes `DCTZ Polynomial D` (voir définition page 18).

Les domaines de booléens conditionnels `ConditionalExtendedBoolean` (abbrégé en `CE`) et de `ConditionalCopyBoolean` (abbrégé en `CCB`) peuvent eux être définis en toutes généralité comme:

```
CE(C,V) : Exports == Implementation where
 C : EBCAT
 V : EBCAT

 Exports ==> Join (CCAT (C,V), EBCAT)
 Implementation ==> CO (C,V)

CCB (C) : Exports == Implementation where

 C : ExtendedBooleanCategory

 V ==> CopyBoolean

 Exports ==> Join(CCAT (C,V),EBCAT)
 Implementation ==> CE (C,V)
```

Les domaines d'anneaux d'objets conditionnels `ConditionalRing` (abbrégé en `CR`) peuvent être définis comme suit:

```
CR (C,V):Exports == Implementation where
```

```
 C : EBCAT
 V : Ring

 Exports ==> Join (CCAT(C,V),Ring)
 Implementation ==> CO (C,V)
```

Les booléens conditionnels dont les conditions sont des `PUIC` sont donc implémentés comme: `CCB (C)`[3],
et les inconnues entières conditionnelles comme: `CR (C,PUI (BasicUnknown,D))`.

3.4 Quelques calculs

3.4.1 Calcul de conditions sur les inconnues entières

```
Starts dribbling to UI.out (1993/12/22, 7:16:25).
(268) ->)read UI

)lo EBCAT EBCAT- EBMACAT EBMACAT- ECCAT DCCAT CTZCAT ECTZCAT
)lo DC DCME ECTZ DCTZ
)lo UCAT URCAT PUI
)lo PFS EPUIC PUIC

   loading ...

Or [positive (a*(b-1))$PUIC ([a,b,c,n],Integer),_
    negative (a*(b-1))$PUIC ([a,b,c,n],Integer),_
    null (a)$PUIC ([a,b,c,n],Integer),_
    null (b-1)$PUIC ([a,b,c,n],Integer)]

   (1)   Or    (a=0)
              (b - 1=0)
             (a b - a<0)
             (a b - a>0)
                                        Type: PUICondition([a,b,c,n],Integer)
simplify expand %

   (2)  True
                                        Type: PUICondition([a,b,c,n],Integer)
And (positive (a*(b-1))$PUIC ([a,b,c,n],Integer),_
     null (a)$PUIC ([a,b,c,n],Integer))

   (3)  (a=0) And (a b - a>0)
                                        Type: PUICondition([a,b,c,n],Integer)
simplify expand %

   (4)  False
                                        Type: PUICondition([a,b,c,n],Integer)
And (null (a*(b-1))$PUIC ([a,b,c,n],Integer),_
     null (b-1)$PUIC ([a,b,c,n],Integer))
```

3. On note `C PUIC (BasicUnknown,D)`.

```
   (5)  (b - 1=0) And (a b - a=0)
                                        Type: PUICondition([a,b,c,n],Integer)
simplify expand %

   (6)   Or (a=0) And (b - 1=0)
                (b - 1=0)
                                        Type: PUICondition([a,b,c,n],Integer)

assert (null(a**2+b**2)$PUIC ([a,b,c,n],Integer),_
        And (null a,null b)$PUIC ([a,b,c,n],Integer))
                2    2
   (7)  [lhs= (b  + a =0),rhs= (a=0) And (b=0)]
                             Type: Record(lhs: PUICondition([a,b,c,n],Integer),
                                          rhs: PUICondition([a,b,c,n],Integer))
Or[positive(a**2+b**2)$PUIC ([a,b,c,n],Integer),_
   negative(a**2+b**2)$PUIC ([a,b,c,n],Integer),_
   null(a)$PUIC ([a,b,c,n],Integer),_
   null(b)$PUIC ([a,b,c,n],Integer)]

   (8)   Or    (a=0)
               (b=0)
             2    2
           (b  + a >0)
                                        Type: PUICondition([a,b,c,n],Integer)
simplify %

   (9)  True
                                        Type: PUICondition([a,b,c,n],Integer)
And (null(a**2+b**2)$PUIC ([a,b,c,n],Integer),_
     null(a)$PUIC ([a,b,c,n],Integer))

                    2    2
   (10)  (a=0) And (b  + a =0)
                                        Type: PUICondition([a,b,c,n],Integer)
simplify %

                    2    2
   (11)  (a=0) And (b  + a =0)
                                        Type: PUICondition([a,b,c,n],Integer)
And (null(a**2+b**2)$PUIC ([a,b,c,n],Integer),_
     positive(b)$PUIC ([a,b,c,n],Integer))

                    2    2
   (12)  (b>0) And (b  + a =0)
                                        Type: PUICondition([a,b,c,n],Integer)
simplify %

   (13)  False
                                        Type: PUICondition([a,b,c,n],Integer)

assert (positive(a**2+b**2)$PUIC ([a,b,c,n],Integer),_
        Not And (null a,null b)$PUIC ([a,b,c,n],Integer))

                2    2
   (14)  [lhs= (b  + a >0),rhs=  Or (a<0)]
                                    (a>0)
                                    (b<0)
                                    (b>0)
```

```
                          Type: Record(lhs:PUICondition([a,b,c,n],Integer),
                                       rhs: PUICondition([a,b,c,n],Integer))
Or (positive(a**2+b**2)$PUIC ([a,b,c,n],Integer),_
    null (a**2+b**2)$PUIC ([a,b,c,n],Integer))

              2    2
   (15)   Or (b  + a =0)
              2    2
            (b  + a >0)
                                       Type: PUICondition([a,b,c,n],Integer)
simplify %

   (16)  True
                                       Type: PUICondition([a,b,c,n],Integer)
assert (positive (2*n+5)$PUIC ([a,b,c,n],Integer),_
        Or(positive(n+2),null(n+2))$PUIC ([a,b,c,n],Integer))

   (17)  [lhs= (2n + 5>0),rhs=  Or (n + 2=0)]
                                   (n + 2>0)
                          Type: Record(lhs:PUICondition([a,b,c,n],Integer),
                                       rhs: PUICondition([a,b,c,n],Integer))
assert (positive (3*n+7)$PUIC ([a,b,c,n],Integer),_
        Or(positive(n+2),null(n+2))$PUIC ([a,b,c,n],Integer))
   (18)  [lhs= (3n + 7>0),rhs=  Or (n + 2=0)]
                                   (n + 2>0)
                          Type: Record(lhs:PUICondition([a,b,c,n],Integer),
                                       rhs: PUICondition([a,b,c,n],Integer))
simplify (Or [positive(n+2)$PUIC ([a,b,c,n],Integer),_
              null(n+2)$PUIC ([a,b,c,n],Integer),_
              negative(2*n+5)$PUIC ([a,b,c,n],Integer),_
              null(2*n+5)$PUIC ([a,b,c,n],Integer)])

   (17)   True
                                       Type: PUICondition([a,b,c,n],Integer)
simplify (Or [positive(2*n+5)$PUIC ([a,b,c,n],Integer),_
              positive(n+2)$PUIC ([a,b,c,n],Integer),_
              null(n+2)$PUIC ([a,b,c,n],Integer)])

    (18)   Or (n + 2=0)
             (n + 2>0)
             (2n + 5>0)
                                       Type: PUICondition([a,b,c,n],Integer)
simplify (Or (positive(2*n+5)$PUIC ([a,b,c,n],Integer),_
              negative(n+2)$PUIC ([a,b,c,n],Integer)))


   (18)   True
                                       Type: PUICondition([a,b,c,n],Integer)
simplify (Or [positive(2*n+5)$PUIC ([a,b,c,n],Integer),_
              negative(3*n+7)$PUIC ([a,b,c,n],Integer),_
              null(3*n+7)$PUIC ([a,b,c,n],Integer)])


   (19) True
                                       Type: PUICondition([a,b,c,n],Integer)
```

3.4.2 Calcul d'inconnues entières conditionnelles

```
)lo PCCAT PC CCAT CO CR CE CF   -- conditional
)lo CB CCB CUICB CUI UICPAC           -- unknownInteger

   loading ...


cu : CUI ([n,m,p],Integer)
                                                                    Type: Void

cu := abs ((n*(n+1)/2)$PUI ([n,m,p],Integer))

                            2
           2              - n  - n
   (20)  ((n  + n<0) -> --------)
                            2
              2
           ((n  + n=0) -> 0)
                           2
            2             n  + n
         ((n  + n>0) -> ------)
                           2
                               Type: ConditionalUnknownInteger([n,m,p],Integer)
expand %


   (21)    ( Or   (n=0)   -> 0)
                (n + 1=0)
                             2
                            n  + n
         ( Or   (n>0)   -> ------)
              (n + 1<0)      2
‘                              Type: ConditionalUnknownInteger([n,m,p],Integer)
(abs cu + abs n)$CUI ([n,m,p],Integer)


                                      2
                      2             - n  - 3n
   (22)  ((n<0) And (n  + n<0) -> ---------)
                                       2
                           2
           ((n<0) And (n  + n=0) -> - n)
                                     2
                        2           n  - n
         ((n<0) And (n  + n>0) -> ------)
                                     2
                           2
            ((n=0) And (n  + n=0) -> 0)
                                     2
                        2           n  + n
         ((n=0) And (n  + n>0) -> ------)
                                     2
                                     2
                        2           n  + 3n
         ((n>0) And (n  + n>0) -> -------)
                                       2
‘                                Type: ConditionalUnknownInteger([n,m,p],Integer)
```

```
expand %

   (23)  ((n<0) And (n + 1=0) -> - n)
                 ((n=0) -> 0)
                          2
                         n  + 3n
              ((n>0) -> -------)
                           2
                           2
                          n  - n
            ((n + 1<0) -> ------)
                           2
‘                             Type: ConditionalUnknownInteger([n,m,p],Integer)
cu := abs cu + cu**4 - cu

                              8     7     6     5    4
                2            n  + 4n  + 6n  + 4n  + n
   (24)  ( Or (n  + n<0) -> -------------------------)
                2                        16
               (n  + n>0)
                          2
                       ((n  + n=0) -> 0)
‘                             Type: ConditionalUnknownInteger([n,m,p],Integer)
expand %

   (25)               ( Or   (n=0)   -> 0)
                           (n + 1=0)
                             8     7     6     5    4
                            n  + 4n  + 6n  + 4n  + n
          ( Or   (n>0)   -> ------------------------)
               (n + 1<0)               16
‘                             Type: ConditionalUnknownInteger([n,m,p],Integer)

And (inf? (cu,0),Not (inf? (cu,0)))$CUICB ([n,m,p],Integer)

   (27)  (True -> False)
                                       Type: CUICopyBoolean([n,m,p],Integer)
Or (inf? (0,cu), inf? (cu,0))$CUICB ([n,m,p],Integer)

                2              8     7     6     5    4
   (28)  ( Or (n  + n<0) And (n  + 4n  + 6n  + 4n  + n =0) -> False)
                                 2
                               (n  + n=0)
                2              8     7     6     5    4
               (n  + n>0) And (n  + 4n  + 6n  + 4n  + n =0)
                2              8     7     6     5    4
          ( Or (n  + n<0) And (n  + 4n  + 6n  + 4n  + n >0) -> True)
                2              8     7     6     5    4
               (n  + n>0) And (n  + 4n  + 6n  + 4n  + n >0)
                                       Type: CUICopyBoolean([n,m,p],Integer)
expand %

   (29)  ( Or   (n=0)   -> False)
               (n + 1=0)
          ( Or   (n>0)   -> True)
               (n + 1<0)
                                       Type: CUICopyBoolean([n,m,p],Integer)
```

Chapitre 4

Conclusion

Les objets conditionnels ainsi que les objets inconnus peuvent être utilisés pour rapprocher la forme des calculs effectués par les systèmes de calcul formel, des calculs faits à la main, et donc faciliter l'accès aux Systèmes de Calcul Formel (voir quelques besoins exprimés par les utilisateurs dans les news 4).
En effet ils permettent d'exprimer des informations qu'il n'est pas possible d'exprimer à l'heure actuelle et ainsi d'aller plus loin dans le calcul symbolique.

On a pu constater que le calcul des objets conditionnels dépend en grande partie du calcul sur les conditions. Dans les cas que nous avons traités, il nous a fallut étudier le signe des expressions puisqu'on avait à analyser des comparaisons à zéro d'expressions. On peut pousser ce travail plus loin, en améliorant le mécanisme d'assertions, ainsi que le simplificateur.

D'autres applications de ces calculs sont prévues en `Axiom` telles que le calcul des limites ...
Mais on peut généraliser l'utilisation de telles expressions à d'autres formes de calcul symbolique tels que la preuve de programme, et en général tout calcul qui comporte des scindages.

Répertoire des abréviations

Voici la liste des foncteurs de catégories et des constructeurs de domaines que nous avons introduits. Pour chaque abréviation, les pages référencées sont celle de la spécification, suivie de celle de l'implémentation (quand elles sont données).

Foncteurs de catégories

CCAT : ConditionalCategory 12,60

CTZCAT : ComparisonsToZeroCategory 17,54

DCCAT : DisjunctiveConditionCategory 11,53

EBCAT : ExtendedBooleanCategory 9,51

EBMACAT : ExtendedBooleanModAssertionsCategory 11,52

ECCAT : ElementaryConditionCategory 11,53

ECTZCAT : ElementaryComparisonsToZeroCategory 18,54

PCCAT : PartialConditionalCategory 12,58

UCAT : UnknownCategory 30,62

URCAT : UnknownRingCategory 31,62

Constructeurs de domaines

CB : CopyBoolean 34

CCB : ConditionalCopyBoolean 37

CE : ConditionalExtendedBoolean 37

CEX : ConditionalExpression 16

CO : Conditional 14

CR : ConditionalRing 37

CUI : ConditionalUnknownInteger 35

CUICB : CUICopyBoolean 34

Constructeurs de packages

Table des matières

Bibliographie

[Abe92] Karl Aberer. Combinatorial models and symbolic computation. In *Proc. of DISCO*, pages 116–131, 1992.

[A.F90] A.Fortenbacher. Efficient type inference and coercion in computer algebra. In A.Miola, editor, *Proc. of DISCO in Capri*, pages 56–60. Springer-Verlag, 1990.

[CAJL91] Ch.Faure, A.Galligo, J.Grimm, and L.Pottier. The extensions of the sisyphe computer algebra system: `ulysse` and `athena`. In *Proc. of DISCO*, 1991.

[Cav70] B.F. Caviness. On canonical forms and simplification. *J. ACM*, 17(2):385–396, April 1970.

[Ch.90] Ch.Faure. A meta simplifier. In *Proc. of ISSAC*, 1990.

[CJ92] C.Lassez and JL.Lassez. Quantifier elimination for conjunctions of linear constraints via convex hull algorithm. In B.R.Donald, D.Kapur, and J.L.Mundy, editors, *Symbolic and Numerical Computation for Artificial Intelligence*, pages 103–119. Academic Press, 1992.

[CR73] C. Chang and R.Lee. *Symbolic Logic and Mechanical Theorem Proving.* Computer Science and Applied Mathematics. Academic Press, Inc., 1973.

[Dav91] J.H. Davenport. The axiom system. 1991.

[Dav92] J.H. Davenport. How does one program in the axiom system. 1992.

[DJ90] N. Dershowitz and J-P. Jouannaud. *Rewrite systems.*, pages 244–320. Elsevier Science Publishers, 1990.

[D.J93] D.J.Jeffrey. Integration to obtain expressions valid on domains of maximal extent. In M.Bronstein, editor, *Proc. of ISSAC in Kiev*, pages 34–41. ACM Press, 1993.

[Fau91] Ch. Faure. Towards a meta simplifier. Technical Report 1402, INRIA, Avril 1991.

[Fau92] Ch. Faure. *Quelques aspects de la Simplification en Calcul Formel.* PhD thesis, Université de Nice Sophia-Antipolis, 1992.

[GGP90] A. Galligo, J. Grimm, and L. Pottier. The design of sisyphe. In *Proc. of DISCO.* Springer Verlag, 1990.

[Gon86] Gaston H. Gonnet. An implementation of operators for symbolic algebra systems. In *Proc. of SYMSAC*, pages 239–243. ACM, 1986.

[Hea68] A. Hearn. Reduce: A user-oriented interactive system for algebraic simplification. In *Interactive Systems for Experimental Applied Mathematics*, pages 79–90. Academic Press, 1968.

[Hea76] Anthony C. Hearn. A new reduce model for algebraic simplification. pages 46–51, 1976.

[Jen88] R.D. Jenks. Scratchpad ii: An abstract datatype system for mathematical computations. In R. Janssen, editor, *Proc. Trends in Computer Algebra*, pages 12–37, 1988.

[J.F94a] Richard J.Fateman. Assumptions, domains, conditional expressions, and types in computer algebra systems – issues and some answers. Submitted to ISSAC'94, 1994.

[JF94b] J.H.Davenport and C. Faure. The "unknown" in computer algebra. *Programmirovanie*, 1994.

[JH84] J-P. Jouannaud and H.Kirchner. Completion of a set of rules modulo a set of equations. In *Proc. 11th ACM Conference of principles of Programming Languages*, 1984. Salt Lake City (Utah, USA).

[JK91] J-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. 1991.

[JL86] J-P. Jouannaud and P. Lescanne. La réécriture. *Technique et Science Informatique*, 5(6):433–452, April 1986.

[JS92a] R.D. Jenks and R.S. Sutor. *Axiom The Scientific Computation System*, chapter 13, pages 527–546. Springer-Verlag, 1992.

[JS92b] R.D. Jenks and R.S. Sutor. *Axiom The Scientific Computation System.* Springer-Verlag, 1992.

[JS92c] R.D. Jenks and R.S. Sutor. *Axiom The Scientific Computation System*, chapter 12, pages 515–526. Springer-Verlag, 1992.

[KB70] D.E. Knuth and P.B. Bendix. *Simple word problems in universal algebras.*, pages 263–297. Pergamon Press, 1970.

[Lan80] D. Lankford. Research in applied equationnal logic. 1980.

[Lon93] H. Long. Quantifier elimination for formulas constrained by quadratic equations. In M.Bronstein, editor, *Proc. of ISSAC in Kiev*, pages 264–274. ACM Press, 1993.

[MJ92] Robert M.Corless and David J.Jeffrey. Well ... it isn't quite that simple. *SIGSAM Bulletin*, 26 No 3, Aug. 1992.

[Mos71] Joel Moses. Algebraic simplification: A guide for the perplexed. *Communications of the ACM*, 14(8):527–537, August 1971.

[PS81] G. Peterson and M. Stickel. Complete sets of reduction for some equational theories. *J. ACM*, 28(2):233–264, 1981.

[TG91] T.Weibel and G.H.Gonnet. An algebra of properties. In S.M.Watt, editor, *Proc. of ISSAC*, pages 352–359. ACM Press, 1991.

Annexe A

extendedBoolean.spad

```
-- File extendedBoolean.spad

)abbrev category EBCAT ExtendedBooleanCategory
)abbrev category EBMACAT ExtendedBooleanModAssertionsCategory
)abbrev category ECCAT ElementaryConditionCategory
)abbrev category DCCAT DisjunctiveConditionCategory
)abbrev category CTZCAT ComparisonsToZeroCategory
)abbrev category ECTZCAT ElementaryComparisonsToZeroCategory
)abbrev domain   DC DisjunctiveCondition
)abbrev domain   DCME DisjunctiveConditionModEquivalences
)abbrev domain   ECTZ ElementaryComparisonsToZero
)abbrev domain   DCTZ DisjunctiveComparisonsToZero
```

```
ExtendedBooleanCategory: Category == SetCategory with
    True : constant -> $
      ++ True is a logical constant.
    False : constant -> $
      ++ False is a logical constant.
    Not : $ -> $
      ++ Not n returns the negation of n.
    And : ($, $) -> $
      ++ And(a,b) returns the logical {\em and} of Boolean \spad{a}
      ++ and b.
    And : List $ -> $
      ++ And [a,b,c] returns the logical {\em and} of a, b and c.
    Or : ($, $) -> $
      ++ Or(a,b) returns the logical inclusive {\em or}
      ++ of Boolean \spad{a} and b.
    Or  : List $ -> $
      ++ Or [a,b,c] returns the logical {\em or} of a, b and c.
    Xor : ($, $) -> $
      ++ Xor(a,b) returns the logical exclusive {\em or}
      ++ of Boolean \spad{a} and b.
    Nand : ($, $) -> $
      ++ Nand(a,b) returns the logical negation of \spad{a} and b.
    Nor : ($, $) -> $
      ++ Nor(a,b) returns the logical negation of \spad{a} or b.
    Implies: ($, $) -> $
      ++ Implies(a,b) returns the logical implication
      ++ of Boolean \spad{a} and b.
    inf? : ($,$) -> Boolean
      ++ inf? (a,b) is true if a is syntactically lower than b
      ++ it can be replaced by some sementical test
    simplify : $ -> $
      ++ simplify (a) ...
 add
   Nand (a,b) == Not And (a,b)
   Nor (a,b) == Not Or (a,b)
   Xor (a,b) == And (Or (a,b),Not And (a,b))
   Implies (a,b) == Or (Not a,b)
   simplify a == a
```

```
ExtendedBooleanModAssertionsCategory A : Category == Exports where

 A : SetCategory

 Exports ==> ExtendedBooleanCategory with
   assertions : -> List A
      ++ assertions () returns the actual list of Equivalences
   asserted : -> $
      ++ asserted () returns the actual extendedboolean equivalent
      ++ to the conjunctions of the assertions
   assert : ($,$) -> A
      ++ assert (a,b) declares that a <=> b
      ++ if a is an elementary condition (E)
      ++ Those Equivalences are used in the check of completude.
   clearAssertions : -> List A
      ++ clearAssertions () suppressed all the Equivalences
   computeModAssertions : $ -> $
      ++ computeModAssertions (a) computes the extendedboolean And (a,
      ++ asserted)
   contradictoryToAssertions : $ -> Boolean
      ++ contradictoryToAssertions (a) is true if And (a, asserted)
      ++ is false

  add
    computeModAssertions x == And (x,asserted)
    contradictoryToAssertions x == computeModAssertions x = False
    simplify x ==
      x = False or x = True => x
      contradictoryToAssertions x => False
      contradictoryToAssertions Not x => True
      computeModAssertions x
```

```
ElementaryConditionCategory : Category == Exports where

 Exports ==> SetCategory with
   inf? : ($,$) -> Boolean
     ++ inf? (a,b) is true if a is syntactically lower than b
   contradictory? : ($,$) -> Boolean
     ++ contradictory? (a,b) is true if (a and b) is False.
   redundant? : ($,$) -> Union (Integer,"failed")
     ++ redundant? (a,b) is -1 if (a and b)<=>a, is 1 if (a and b)<=>b
     ++ is 0 if a=b or is "failed"
   Not : $ -> List List $
     ++ Not (a) builds the condition opposite to a in a
     ++ disjunctive form.

DisjunctiveConditionCategory E : Category == Exports where

 E : ElementaryConditionCategory

 Conjunction ==> List E           -- implicit 'and' [] = true
 Disjunction ==> List Conjunction -- implicit 'or'  [] = false

 Exports ==> ExtendedBooleanCategory with
   destruct     : $ -> Disjunction
     ++ destruct (a)
   construct    : E -> $
     ++ construct (e)
   construct    : Disjunction -> $
     ++ construct (disj)
```

```
ComparisonsToZeroCategory V : Category == Implementation where

 V : SetCategory

 Implementation ==> SetCategory with
   positive : V -> $
     ++ positive (v) builts an expression which will be evaluated in
     ++ true if v is evaluated in a positive value.
   negative : V -> $
     ++ negative (v) builts an expression which will be evaluated in
     ++ true if v is evaluated in a negative value.
   null : V -> $
     ++ null (v) builts an expression which will be evaluated in
     ++ true if v is evaluated in a null value.

ElementaryComparisonsToZeroCategory V : Category == Exports where

 V : SetCategory

 Exports ==> Join(ElementaryConditionCategory,
                  ComparisonsToZeroCategory V)
  with
   value : $ -> V
     ++ value (a) gives the value compared to zero
   sign  : $ -> SmallInteger
     ++ sign (a) gives the sign given to the value
   element : (V,SmallInteger) -> $
     ++ element (v,opv) builds the elementary condition [v,opv]
   infV? : (V,V) -> Boolean
     ++ infV? (v1,v2) is true if v1 < v2
     -- If this operation is not exported, we can't have multiple
```

```
DisjunctiveConditionModEquivalences E: Exports == Implementation where

 E : ElementaryConditionCategory

 Equivalence ==> Record (lhs:$,rhs:$)

 Exports ==> Join (DisjunctiveConditionCategory E,
                   ExtendedBooleanModAssertionsCategory Equivalence)

 Implementation ==> DisjunctiveCondition E add
   Rep := DisjunctiveCondition E

   ASSERTIONS : List Equivalence
   ASSERTIONS := empty
   ASSERTED : $
   ASSERTED := True

   assertions () == ASSERTIONS
   asserted () == ASSERTED
   clearAssertions () ==
     ASSERTED := True
     ASSERTIONS := empty

   equi (r1:Rep,r2:Rep):Rep ==
     And$Rep (Implies$Rep (r1,r2),Implies$Rep (r2,r1))

   assert (x1,x2) ==
     x1 ^= x2 =>
       assert := asserted
       aux := computeModAssertions equi (x1::Rep,x2::Rep)
       aux ^= assert =>
         aux = False$Rep =>
           error "This assertion is contradictory with those previously asserted"
         aux ^= True$Rep =>
           ASSERTED := aux
           ass := [x1,x2]
           ASSERTIONS := concat (assertions (),ass)
           ass
         [x1,x2]
       [x1,x2]
     [x1,x2]

   simplifyToTrueOrFalse (x:$):$ ==
     contradictoryToAssertions x => False
     contradictoryToAssertions Not x => True
     x

   simplify x ==
     x = False or x = True => x
     simplifyToTrueOrFalse x
```

```
DisjunctiveComparisonsToZero (E,V): Exports == Implementation where

 V : SetCategory
 E : ElementaryComparisonsToZeroCategory V

 D ==> DisjunctiveConditionModEquivalences E
 Equivalence ==> Record (lhs:$,rhs:$)

 Exports ==> Join (DisjunctiveConditionCategory E,
                   ExtendedBooleanModAssertionsCategory Equivalence,
                   ComparisonsToZeroCategory V)

 Implementation ==> D add
   Rep := D

   Conj := List E
   Disj := List List E

   trueD? : Disj -> Boolean

   termInConj (v:V,opv:SmallInteger,c:Conj):Integer ==
      position (element (v,opv),c)
   restOfConj (c:Conj,li:List Integer):Conj ==
      res := c
      for i in li repeat
          res := remove (qelt (c,i),res)
      res
   termInConj? (v:V,opv:SmallInteger,c:Conj):Boolean ==
      termInConj (v,opv,c) ^= 0
   valueInConj? (v:V,c:Conj):Boolean ==
      termInConj? (v,-1,c) or termInConj? (v,0,c) or termInConj? (v,1,c)

   others (v:V,d:Disj):Disj ==
      [c for c in d| not valueInConj? (v,c)]

   trueLD? (v:V,d:Disj,ld:List Disj):Boolean ==
      and/[ld.first = d1 for d1 in ld.rest] =>
          trueD? append (ld.first,others (v,d))
      and/[trueD? d1 for d1 in ld] and trueD? others (v,d)

   factorizeV (v:V,opv:SmallInteger,d:Disj):Union(Disj,"failed") ==
      sign : Disj
      sign := []
      sign? := false
      for c in d repeat
          i := termInConj (v,opv,c)
          i ^= 0 =>
            sign? := true
            not empty? c.rest => sign := concat (sign,restOfConj (c,[i]))
      not sign? => "failed"
      sign

   syntacticallyTrue? (v:V,d:Disj):Boolean ==
      neg := factorizeV (v,-1,d)
      nul := factorizeV (v,0,d)
      pos := factorizeV (v,1,d)
      neg case "failed" or nul case "failed" or pos case "failed" =>
          false
```

```
      trueLD? (v,d,[neg::Disj,nul::Disj,pos::Disj])

   trueD? d ==
      empty? d => true
      t := d.first.first
      syntacticallyTrue? (value t,d)

   simplifyToTrue (x1:$):$ ==
      x1 = False or x1 = True => x1
      trueD? destruct x1 => True
      x1

   simplify x1 == simplify$Rep (simplifyToTrue x1)::Rep

   assert (x1,x2) == assert$Rep (x1,x2)
   assertions == assertions$Rep
   clearAssertions == clearAssertions$Rep

   positive v == construct positive$E v
   negative v == construct negative$E v
   null v     == construct null$E v
```

Annexe B

conditional.spad

```
-- File conditional.spad

)abbrev category PCCAT PartialConditionalCategory
)abbrev package PCF PartialConditionalFunctions

)abbrev category CCAT ConditionalCategory
)abbrev package CF ConditionalFunctions



PartialConditionalCategory (C:ExtendedBooleanCategory,
                            V:SetCategory):Category == SetCategory with
  empty : -> $
   ++ empty () builts an empty conditional object
  empty? : $ -> Boolean
   ++ empty? (a) is yes if a = empty ()
  firstCond : $ -> C
   ++ firstCond (a) gives the first condition of a
  firstVal : $ -> V
   ++ firstVal (a) gives the first value of a
  rest : $ -> $
   ++ rest (a) gives a except the first element
  concat : ($,$) -> $
   ++ concat (a,b) adds all the element of a and all the element
   ++ of b and returns "failed" if the result isn't coherent
  concat : ($,C,V) -> $
   ++ concat (a,c,v) adds the couple (c,v) at the end of a
   ++ if c is not False and returns "failed" if the result isn't
   ++ coherent
  elt : (C,V) -> $
   ++ elt (c,v) builts [[c,v]]
  conditions : $ -> List C
   ++ conditions (C) gives the list of all the conditions
  values : $ -> List V
   ++ values (a) gives the list of all the values
```

```
PartialConditionalFunctions (C,V1,V2) : Exports == Implementation where

 C : ExtendedBooleanCategory
 V1 : SetCategory
 V2 : SetCategory

 PCV1 ==> PartialConditional (C,V1)
 PCV2 ==> PartialConditional (C,V2)

 Exports ==> with
  apply1 : ((C,V1) -> PCV2,PCV1) -> PCV2
   ++ apply1 (fun,x) applies the function fun to each couple (c,v)
   ++ of x
  apply2 : ((C,V1,C,V1) -> PCV2,PCV1,PCV1) -> PCV2
   ++ apply2 (fun,x,y) applies the function fun to each n-uplet
   ++ (c1,v1,c2,v2) where (c1,v1) comes from x and (c2,v2) comes
   ++ from y

 Implementation ==> add
   apply1 (fun,x1) ==
      pcv := empty$PCV2
      while not empty? x1 repeat
            pcv := concat (pcv,fun(firstCond x1,firstVal x1))
            x1 := rest x1
      pcv

   apply2 (fun,x1,x2) ==
      pcv := empty$PCV2
      while not empty? x1 repeat
            aux2 := x2
            while not empty? aux2 repeat
                  pcv := concat (pcv, fun(firstCond x1,firstVal x1,
                                          firstCond aux2,firstVal aux2))
                  aux2 := rest aux2
            x1 := rest x1
      pcv
```

```
ConditionalCategory (C:ExtendedBooleanCategory,
                     V:SetCategory): Category == SetCategory with

   apply1 : (V -> V,$) -> $
   ++ apply1 (fun,a) applies fun to every value in a and built
   ++ the conditional object equivalent to [[c1,fun a1],[c2,fun a2] ...]
   ++ if a = [[c1,v1],[c2,v2],...]
   apply2 : ((V,V) -> V,$,$) -> $
   ++ apply2 (fun,a,b) applies fun to every couple of values (ai,bj)
   ++ where ai is in a and bj in b and builts the conditional object
   ++ equivalent to [[And(cai,cbj),fun(ai,bj)] ...]
   conditionalValue : (C,$,$) -> $
   ++ conditionalValue (c,a,b) computes the conditional object
   ++ equivalent to [[c,a],[Not c,b]]
   values : $ -> List V
   ++ values (a) gives all the possible values for the expression a
   conditions : $ -> List C
   ++ conditions (a) returns the list of conditions of the conditional
   ++ expression
   whichCondition : ($,V) -> C
   ++ whichCondition (a,v) gives the condition under which the expression
   ++ a reach the value v
   whichValue : ($,C) -> V
   ++ whichValue (a,c) gives the value reached by a under the condition
   ++ c
   setelt : ($,C,$) -> $
   ++ setelt (a,c,b) replaces the values of a by those in b when
   ++ c belongs to the conditions of a
   RetractableTo V
   ++ coerce (v) returns v as an element of $.
   ++ retract (a) returns the first value of a
   ++ retractIfCan (a) returns v if a is True -> v
   coerce : PartialConditional (C,V) -> $
   ++ coerce (pcv) computes the complete conditional object equivalent
   ++ to pcv if pcv is complete
   coerce : $ -> PartialConditional (C,V)
   ++ coerce (a) computes the partial conditional object equivalent
   ++ to a
```

```
ConditionalFunctions (C,V1,V2,CV1,CV2) : Exports == Implementation where

 C   : ExtendedBooleanCategory
 V1  : SetCategory
 V2  : SetCategory
 CV1 : ConditionalCategory (C,V1)
 CV2 : ConditionalCategory (C,V2)

 PCV1 ==> PartialConditional (C,V1)
 PCV2 ==> PartialConditional (C,V2)

 Exports ==> with
   apply1 : (V1 -> V2,CV1) -> CV2
   ++ apply1 (fun,a) applies fun to every value in a and built
   ++ the conditional object equivalent to [[c1,fun a1],[c2,fun a2] ...]
   ++ if a = [[c1,v1],[c2,v2],...]
   apply2 : ((V1,V1) -> V2,CV1,CV1) -> CV2
   ++ apply2 (fun,a,b) applies fun to every couple of values (ai,bj)
   ++ where ai is in a and bj in b and builts the conditional object
   ++ equivalent to [[And(cai,cbj),fun(ai,bj)] ...]

 Implementation ==> add
   apply1 (fun,x1) ==
      pcv := empty$PCV2
      aux1 := x1::PCV1
      while not empty? aux1 repeat
            pcv := concat (pcv,firstCond aux1,fun firstVal aux1)
            aux1 := rest aux1
      pcv::CV2

   apply2 (fun,x1,x2) ==
      pcv := empty$PCV2
      aux1 := x1::PCV1
      aux2 := x2::PCV1
      while not empty? aux1 repeat
            aux := aux2
            while not empty? aux repeat
                  cond := simplify And (firstCond aux1,firstCond aux)
                  pcv := concat (pcv,cond,fun (firstVal aux1,firstVal aux))
                  aux := rest aux
            aux1 := rest aux1
      pcv::CV2
```

Annexe C

unknown.spad

```
-- File unknown.spad

)abbrev category UCAT UnknownCategory
)abbrev category URCAT UnknownRingCategory



UnknownCategory (OrderedBasicUnknown):Category
   == Exports where

 OrderedBasicUnknown : OrderedFinite

 Exports ==> SetCategory with
    RetractableTo OrderedBasicUnknown
    RetractableTo Symbol

UnknownRingCategory (OrderedBasicUnknown,D):Category
   == Exports where

 D:Ring
 OrderedBasicUnknown:OrderedFinite

 Exports ==> Join(UnknownCategory OrderedBasicUnknown,Ring) with
    retractIfCan : Polynomial D -> Union ($,"failed")
      ++ retractIfCan (p) computes the object equivalent to p
      ++ if it's possible
```

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399